Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 50 (2008) 1003–1034
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

A catalog of architectural primitives for modeling architectural patterns

Uwe Zdun [a,*], Paris Avgeriou [b]

[a] Distributed Systems Group, Vienna University of Technology, Austria
[b] Department of Computer Science, University of Groningen, The Netherlands

Received 18 May 2007; received in revised form 11 September 2007; accepted 25 September 2007
Available online 11 October 2007

Abstract

Architectural patterns are a fundamental aspect of the architecting process and subsequently the architectural documentation. Unfortunately, there is only poor support for modeling architectural patterns for two reasons. First, patterns describe recurring design solutions and hence do not directly match the elements in modeling languages. Second, they support an inherent variability in the solution space that is hard to model using a single modeling solution. This paper proposes to address this problem by finding and representing architectural primitives: fundamental, formalized modeling elements in representing patterns. In particular, we examined architectural patterns from the components and connectors architectural view, and we discovered recurring primitive abstractions among the patterns, that also demonstrate a degree of variability for each pattern. We used UML 2 as the language for representing these primitive abstractions as extensions of the standard UML elements. The contribution of this approach is that we provide a generic and extensible concept for modeling architectural patterns by means of architectural primitives. Also, we can demonstrate a first set of primitives that participate in several well-known architectural patterns.

Keywords: Software architecture; Software patterns; Pattern primitives; Modeling; UML

1. Motivation

The software architecture of a system needs to be rigorously documented in order to profit from the advantages of architecture-centric development and evolution. One of the most significant aspects of documenting software architectures is the representation of architectural patterns (also known as architectural styles)[1]. In general, a pattern is a problem-solution pair in a given context. A pattern does not only document 'how' a solution solves a problem but also 'why' it is solved, i.e., the rationale behind this particular solution. Architectural patterns help to document architectural design decisions, facilitate communication between stakeholders through a common vocabulary, and assist in analyzing the quality attributes of a software system.

There are three major approaches, that have been used so far for modeling architectural patterns:

1. Architecture Description Languages (ADLs), which aim at representing software architectures in general [27];
2. the Unified Modeling Language which is a generic modeling language but can also be used to describe software architectures [37,26,4];
3. some formal or semi-formal approaches for the formalization of pattern specifications [8,29,43,25].

Unfortunately, none of these approaches succeeds in effectively modeling architectural patterns for the following reasons:

* Corresponding author.
E-mail addresses: zdun@infosys.tuwien.ac.at (U. Zdun), paris@cs.rug.nl (P. Avgeriou).

[1] In this paper we do not distinguish between the terms 'architectural pattern' (used e.g. in [6,39,44]) and 'architectural style' (used e.g. in [42]). For the sake of simplicity, we shall use only the term 'architectural pattern' for the rest of this paper. Their commonalities and differences are elaborated in [2].

0950-5849/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2007.09.003

- The approaches are too limited in the abstractions they propose to grasp the rich concepts found in patterns. UML, to start with, falls short in offering certain standard concepts of architectural patterns [1,26,20]. For example in the 'pipes and filters' architectural pattern [42,6], a pipe does not match the UML connector, since the latter cannot have an associated state or even interfaces. Furthermore there are no elements in UML to model architectural configurations such as a virtual machine [42], a blackboard [3], or a C2 topology [26]. In contrast, many ADLs inherently support a few specific patterns such as C2 [26] or *pipes and filters* [42,6], or can be extended to represent patterns (e.g. using style repositories [30]). But except for these few patterns, ADLs do not support the rest of the patterns. Similarly, the third aforementioned approach is basically concerned with just a handful of design patterns from [11].
- The approaches do not deal with the inherent variability of architectural patterns. This is not restricted to architectural patterns but it is a general problem of specifying patterns because each pattern covers not only one (parametric) solution, but informally describes a whole solution space for a recurring design problem. It is obvious in UML and ADLs, and even more so in the third aforementioned approach that deals with the formal specification of design patterns [8,29,43,25]: such methods are capable of specifying one particular solution in the solution space of the pattern, but fail to specify the whole solution space covered by the informal pattern description.

We propose to remedy the problem of modeling architectural patterns through identifying and representing a number of 'architectural primitives' that can act as the participants in the solution that patterns convey. We use the term 'primitive' because they are the fundamental modeling elements in representing a pattern, and they are the smallest units that makes sense at the architectural level of abstraction (e.g. specialized components, connectors, ports, interfaces). Our approach relies on the assumption that architectural patterns contain a number of architectural primitives that are recurring participants in several other patterns [28]. These primitives are common among the different patterns even if their semantics demonstrate slight variations from pattern to pattern. We have 'mined' a number of architectural patterns and discovered several architectural primitives that we believe are key concepts in modeling architectural patterns and subsequently software architectures in general. We provide a modeling abstraction for each type of elicited architectural primitive, and then demonstrate that it is possible to model architectural patterns explicitly, precisely, and intuitively, through a case study. It is noted that the set of primitives identified in this paper is not exhaustive, but does contain some of the most common primitives found in popular architectural patterns.

Our general approach to define architectural primitives can take advantage of any modeling language, as long as it can be extended to provide the syntax and semantics of the primitives. We have chosen the Unified Modeling Language for this purpose, because it has become the 'lingua franca' of software design and is vastly supported by tools. We have specified an extension of UML 2.0 metaclasses for each elicited primitive, using the standard UML extension mechanisms: stereotypes, tag definitions, and constraints. We have also used the Object Constraint Language (OCL) to formalize the constraints and provide more precise semantics of the primitives. The result is a UML profile that can be imported in modeling tools; in our case we specified the profile in Eclipse/Octopus. We have also developed a model validator as a prototype implementation for supporting model-driven development using our concepts.

The rest of this paper is structured as follows: In Section 2 we give an overview of the proposed approach. Section 3 presents the UML extension mechanism of 'Profiles' and the subset of the UML 2.0 meta-model that was used for specifying our Profile. Section 4 elaborates on the results of the approach by demonstrating several architectural primitives that were mined from some of the most popular architectural patterns. Section 5 demonstrates the approach through a case study, while Section 6 further presents a prototype tool that validates the proposed architectural primitives in a model-driven development context. Finally, Section 8 discusses related work in this field, and Section 9 sums up with conclusions and future work.

2. The proposed approach

The underlying idea behind our approach is that the various architectural patterns share some common architectural 'primitives'. Thus we use the patterns as a foundation to elicit the recurring architectural primitives for a particular architectural view. Specifically, we propose the following approach:

1. Analyze the architectural patterns of a given architectural view to discover common participants in their solutions. These should be recurring and probably varying instances of the same architectural concept, e.g. a special-purpose component or connector. Patterns (a) capture the variations of a solution and (b) describe the solution in a realization-independent way. For instance, pattern descriptions contain pattern variants, implementation hints, design alternatives, consequences, forces that govern a solution, and so forth. These are all sources for eliciting the architectural primitives.
2. Model these architectural primitives as extensions of UML. First we find the UML metaclasses that are a close semantic match to the primitives, e.g. components, connectors, interfaces etc. Then define the semantics of these primitives more precisely with the help of OCL in order to facilitate the unambiguous and consistent modeling of patterns.

3. Use the derived UML extensions of primitives to model pattern instances in real case studies and validate the effectiveness of the primitive to unambiguously model architectural patterns (e.g. using tool support).

It is noted that the pool of architectural patterns, we used to elicit primitives, includes some patterns that are described as 'design patterns' in the literature. In general it is hard to draw the line between architectural patterns and design patterns. In fact, it depends heavily on the viewpoint of the designer or architect whether a specific pattern is categorized as an architectural pattern or a design pattern. Consider for instance, a classical design pattern, the INTERPRETER [11]. The description in [11] presents it as a concrete design guideline. Yet, instances of the pattern are often seen as a central elements in the architecture of software systems, because an INTERPRETER is a central, externally visible component – i.e., the pattern is treated like an architectural pattern (see [42]). Thus, in this paper, we refer to such design patterns as architectural patterns, considering them at an architectural level of abstraction. However, this has resulted in few object-oriented concepts being used in the primitives, e.g. composition and aggregation cascade use object-oriented inheritance.

3. Extending UML to represent the primitives

3.1. A UML profile

According to the UML standard there are two ways to extend the language: the *hard extension* produces an extension of the language meta-model, i.e., a new member of the UML family of languages is specified; the *soft extension* results in a *profile*, which is a set of stereotypes, tag definitions, and constraints that are based on existing UML elements with some extra semantics according to a specific domain. In order to model the architectural primitives we chose the *soft* extension mechanism of UML, i.e., the definition of a profile for architectural primitives for the following reasons:

- A UML profile is good enough for this task since there are already existing UML metaclasses that are semantically a close match to the architectural primitives. Therefore we can simply extend the semantics of these metaclasses rather than having to define completely new metaclasses.
- The users of this profile will feel comfortable by using stereotypes that are extensions of existing metaclasses rather than using concepts they are not familiar with. The learning curve can thus be minimized.
- A profile is still valid, standard UML, so we can count on support from the existing UML tools, rather than offer proprietary UML tools which are rarely used in practice.

We also use OCL to define the necessary constraints for the defined stereotypes to formalize their semantics. OCL constraints are the primary mechanism for traversing UML models and specifying precise semantics on metaclasses and stereotypes.

3.2. The UML 2 meta-model

This section briefly presents part of the existing UML 2.0 meta-model for architectural description, and in particular those metaclasses that we have extended to model the architectural primitives. It is noted that, according to the software architecture community, an architectural description is comprised of multiple views [7,18,19,23]. In this paper we focus on the view that is considered to contain the most significant architectural information, which is the *component-and-connector* view [7]. This view deals with the components, which are units of runtime computation or data-storage, and the connectors which are the interaction mechanisms between components [34,7]. We have focused on this view because the patterns that we have mined concern mainly this view. However, other architectural patterns from other views, such as the 'logical' or 'module' view, can also be searched for primitives, as will be stated in Section 9.

The following UML 2.0 metaclasses are extended to model architectural primitives in the component and connector view, mainly taken from the composite structures and components packages:

1. *Components* are specializations of classes and therefore have attributes and operations, but are also associated with provided and required interfaces. Finally components inherit indirectly from EncapsulatedClassifier and thus may own ports that formalize their interactions points.
2. *Interfaces* serve as contracts that components must comply with. An interface is either a *provided interface* that describes a set of functionalities offered by a component, or a *required interface* that describes a set of functionalities that a component expects from its environment.
3. *Ports* specify a distinct interaction point between the component that owns the port and its environment, or between the component and its internal parts (properties). Ports may specify required and provided interfaces of the component that owns them.
4. *Connectors* are either *assembly connectors* that connect the required interface of one component to the provided interface of a second, or *delegation connectors* that link the ports of a component to its internal parts.
5. *Packages* are mechanisms for grouping model elements either by owning them or importing them. They also provide a namespace for uniquely identifying the elements by their name.

We have also used the following UML metaclasses in order to express the OCL constraints while traversing the UML meta-model: AggregationKind, Association, Classifier, ConnectableElement, ConnectorEnd, EncapsulatedClassifier, Feature, RedefinableElement, Namespace,

NamedElement, PackageableElement, Property, RedefinableElement, VisibilityKind.

It is noted that UML 2.0 provides the means to describe design patterns through the Collaboration metaclass, as an interaction between instances of components and connectors. However, we do not use this metaclass since it is also bounded by the limitations for modeling patterns discussed in Section 1.

The specification of the primitives was implemented with the help of the Octopus plug-in (http://www.klasse.nl/) in the Eclipse environment (http://www.eclipse.org/). We chose this tool for specifying the primitives because Eclipse is open-source and widely used, and also because the Octopus plug-in can statically check OCL 2.0 constraints. For all OCL constraints we assume the standard UML 2.0 role names for the extensions: "base$X", where $X is the extended metaclass, and "extension$Y", where $Y is the stereotype name. Additionally, we have also implemented our own model validator tool to support model-driven development using our concepts (see Section 6 for details).

Fig. 1 illustrates the part of the existing UML meta-model that contains the aforementioned metaclasses and shows their relationships, especially for traversing OCL constraints. The figure has been adapted from the UML 2 standard [32] and, for simplicity, some details have been omitted.

4. Modeling architectural primitives

In this section, we provide more details about our approach, demonstrating the elicitation of architectural primitives from general-purpose architectural patterns, and modeling them with a UML 2.0 profile. We first show the template for documenting the architectural primitives and continue with an elaborate presentation of nine primitives.

4.1. Template for architectural primitive documentation

We propose a simple template for documenting the elicited architectural primitives, consisting of four elements:

- *Introduction*: A brief textual description and discussion of the architectural primitive.
- *Known uses in patterns*: A short description of the patterns in which the architectural primitive participates.

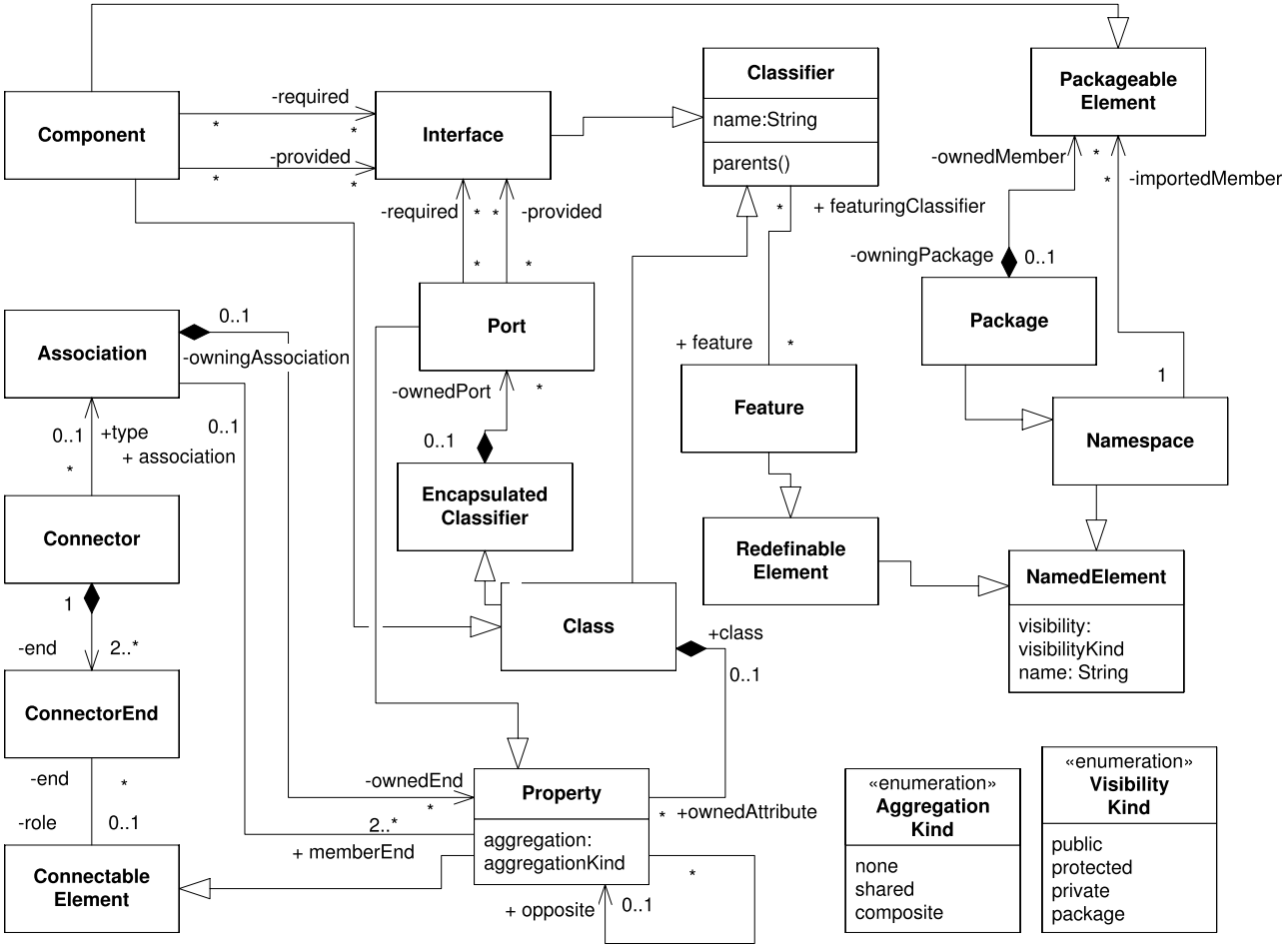

Fig. 1. Part of the UML 2.0 meta-model that was used for the stereotype definition.

- *Modeling issues*: An explanation why this primitive cannot be modeled with standard UML and thus needs to be supported with a UML extension.
- *Modeling solution*: A description of UML 2.0 extensions, containing stereotypes, possibly with tag definitions, and constraints.

4.2. Callback

4.2.1. Introduction

'Callback' is described as follows:

> A callback denotes an invocation to a component B that is stored as an invocation reference in a component A. The callback invocation is executed later, upon a specified set of runtime events, usually implemented as methods. Between two components A and B, a set of callbacks can be defined, also usually implemented as methods. Note that in this description A might be equal to B. In essence, the callbacks between two components A and B are a set of tuples. Each tuple consists of one method $methodA_x \in Methods_A$ that represents a trigger event and a method $methodB_x \in Methods_B$ that is a callback, like:

$$\begin{aligned} Callbacks_{AB} = \{&(methodA_2, methodB_1),\\ &(methodA_1, methodB_2),\\ &(methodA_2, methodB_3),\\ &\ldots\} \end{aligned}$$

> There are two main variants of callbacks:
>
> - The runtime events are ordinary method invocations, field accesses, or other events in the program flow. (Note that these are also called 'joinpoints' in aspect-oriented programming [22]).
> - The runtime events are 'real events' in an event-based programming system, triggered by some event loop.
>
> With regard to modeling the callback, the two variants make no difference: Structurally, both kinds of callbacks are realized in the same way. Sometimes a callback has only one associated runtime event (e.g. a set with only one tuple), sometimes it is raised by a number of different runtime events.

4.2.2. Known uses in patterns

- In the OBSERVER pattern [11] an observer component is notified by one or more subjects about state changes and other events. Usually the notification is realized as a callback.
- MODEL-VIEW-CONTROLLER [6] uses callbacks to inform views about changes in the model, much like the logic behind the OBSERVER pattern.
- A REACTOR [39] is a special kind of OBSERVER that is informed about network events using callbacks.
- In the EVENT SYSTEM pattern [42] components may broadcast a number of events. Another component may register an interest in an event by associating a callback with the event. When an event occurs, the EVENT SYSTEM dispatches all the callbacks associated with the event.
- There are various patterns describing interception architectures, such as INTERCEPTOR [39], MESSAGE INTERCEPTOR [45], and INVOCATION INTERCEPTOR [44]. Interceptors are invoked as extensions to some other invocation; thus they must be invoked, when this other invocation takes place. Usually, the interceptors are triggered by callback events like 'invocation arrived' or 'invocation finished'.
- VISITORS [11] are used to define an interpretation mechanism apart from the structure to be interpreted. They are usually called back, by the elements to be visited.

4.2.3. Modeling issues

A major problem in modeling these patterns in UML is that, even though the callback-structure is a key participant in the patterns, it cannot be explicitly modeled and made visible in UML diagrams, such as component diagrams, class diagrams, or sequence diagrams. There are only some 'hints' that might imply the presence of a callback but there is much ambiguity that could lead to false detections of callbacks. Consider the following examples of such 'hints':

- A structural indicator for a callback that we could include in UMLs structural diagrams is to have a class or a component A store a reference to a method of B. Using this indicator, however, is problematic because there is no unambiguous indication whether the method reference is intended for being used as a callback or not. To make matters worse, invocation references are not necessarily realized by using a reference to a method. Many programming languages do not require a reference to the callback operation at all. For instance, in Java it is sufficient to have the operation name stored in a string to be able to look-up the operation using reflection. When the pattern COMMAND [11] is used, the callback can be encapsulated in the COMMAND. In both cases, the intended use of these structures as callbacks is not directly visible in a UML model.
- Another structural hint for callbacks is their return type. In event-driven applications, the return type of a callback is usually `void`, because the callback is raised by an event, and thus the callback cannot return anything. However, this is not always the case: For instance, an interceptor often returns an error state to indicate to the interceptor architecture, whether the interceptor invocation was successful or not. Also, in non-event-driven applications, for instance, in the VISITOR and OBSERVER patterns, this rule-of-thumb does not hold: Here, the callback may well be used with a return value.

- In some cases, where the callback can be modeled as simple recursive invocations (as in the VISITOR pattern), we can get around this problem by using an accompanying sequence diagram that shows the recursive callback (e.g. class *A* calls *B* and then *B* calls *A* back). However, there are two basic problems with this approach:
  - *No semantic annotation*: Even though the sequence diagram has a callback-like structure, the same kind of sequence diagram might be used for a 'normal' invocation going back and forth, which is not a callback.
  - *Temporal decoupling*: Callbacks are usually stored until an event happens, often much later in time, and then they are invoked upon that event. This cannot be easily depicted with a sequence diagram because of the many invocations that happen between performing the callback and the event that caused it to be invoked.

In summary, UML elements can be used as an indicator that a callback is used, but the callback-structure cannot be identified unambiguously in UMLs structural and interaction diagrams. Thus, the runtime behavior and interaction semantics of the callback-structure cannot be properly modeled in standard UML.

4.2.4. Modeling solution

To capture the semantics of callbacks properly in UML and tackle the above problems, we propose five new stereotypes:

- «*IEvent*»: A stereotype that extends the 'Interface' metaclass and contains a number of methods that are exclusively trigger events for a callback.
- «*ICallback*»: A stereotype that extends the 'Interface' metaclass and contains a number of methods that serve exclusively as callback methods.
- «*EventPort*»: A stereotype that extends the 'Port' metaclass and is typed by two interfaces: IEvent as a *provided* interface and ICallback as a *required* interface. This can be formalized using two OCL constraints for EventPort:

```
-- An event port is typed by IEvent as a
-- provided interface
inv: self.basePort.required->size()=1
  and self.basePort.required->forAll(
    i:Core::Interface|
      ICallback.baseInterface->
        exists(j|j=i))


-- And: An event port is typed by ICallback
-- as a required interface.
inv: self.basePort.provided->size()=1
  and self.basePort.provided->forAll(
    i:Core::Interface|
      IEvent.baseInterface->exists(j|j=i))
```

- «*CallbackPort*»: A stereotype that extends the 'Port' metaclass and is typed by two interfaces: ICallback as a *provided* interface and IEvent as a *required* interface. This can be formalized using two OCL constraints for CallbackPort:

```
-- A callback port is typed by ICallback as a
-- provided interface
inv: self.basePort.required->size()=1
  and self.basePort.required->forAll(
    i:Core::Interface|
      IEvent.baseInterface->exists(j|j=i))


-- And: A callback port is typed by IEvent
-- as a required interface.
inv: self.basePort.required->size()=1
  and self.basePort.required->forAll(
    i:Core::Interface|
      ICallback.baseInterface->
        exists(j|j=i))
```

- «*Callback*»: A stereotype that extends the 'Connector' metaclass and specifies the semantics of a callback connector, which connects an EventPort of a component to a matching CallbackPort of another component. This can be formalized using two OCL constraints:

```
-- A Callback connector has only two ends.
inv: self.baseConnector.end->size()=2

-- A Callback connector connects an EventPort
-- of a component to a matching CallbackPort
-- of another component. An EventPort matches
-- a CallbackPort if the provided IEvent
-- interface of the former matches the
-- required IEvent interface of the latter,
-- and the required ICallback interface of
-- the former matches the provided ICallback
   interface of the latter:
inv: self.baseConnector.end->forAll(
  el,e2:Core::ConnectorEnd|el<>e2 implies(
    (el.role->notEmpty() and
    e2.role->notEmpty()) and
    (if EventPort.basePort->exists(p|
      p.oclAsType(Core:: ConnectableElement)=
      el.role)
    then
      (CallbackPort.basePort->exists(p|
        p.oclAsType(
        Core::ConnectableElement)=
        e2.role)
      and
      el.role.oclAsType (Core::Port).required=
      e2.role.oclAsType (Core::Port).provided
      and
      el.role.oclAsType (Core::Port).provided=
      e2.role.oclAsType (Core::Port).required)
    else
      CallbackPort.basePort->exists(p|
        p.oclAsType(
        Core::ConnectableElement)=
```

```
        el.role)
    endif)))
```

Fig. 2 illustrates these stereotypes according to the UML 2.0 Profiles package, while Fig. 3 depicts the notation used for the stereotypes. All stereotypes use the notation of the metaclass they extend adorned by the name of the stereotype in guillemets.

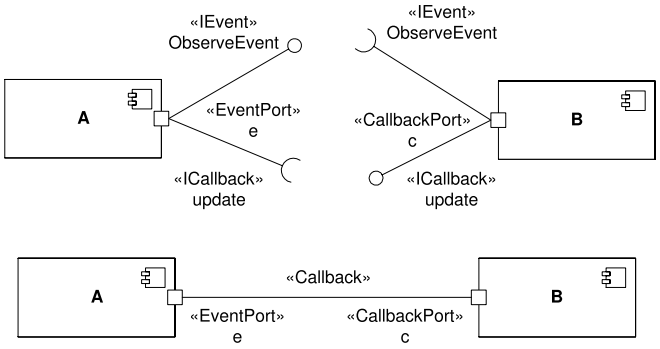

Fig. 3. The notation of the stereotypes in Callback modeling.

4.3. Indirection

4.3.1. Introduction

Indirection happens when one or more related "proxy" components receive a message on behalf of one or more "target" components and forward the message to these "targets", perhaps after some computation has taken place. Afterward the result is sent back, again through the "proxy" components.

Indirection can take place at small scale, with only one client, one proxy, and one target component. It can also involve multiple components playing the role of any of these participants. For instance, a whole layer or subsystem, consisting of multiple components and connectors, might indirect invocations to other components.

4.3.2. Known uses in patterns

- An INDIRECTION LAYER [45] is a general pattern describing a LAYER [6] that redirects all invocations from one system context into another.
- Ordinary LAYERS [6] redirect invocations from layer X to the layer beneath, X − 1.
- A VIRTUAL MACHINE [42] redirects invocations from a byte-code layer into an implementation layer for the commands of the byte-code.
- An INTERPRETER [42,11] redirects invocations from a script (interpreted code) layer into an implementation layer for the commands of the script (interpreted code).
- An ADAPTER [11] redirects invocations from one interface to another.
- FACADE [11] shields a subsystem and redirects invocations into that subsystem.
- A PROXY [11] is a placeholder of another object and redirects invocations to that object.
- A CLIENT PROXY [44] is a special PROXY in the distributed system context.
- A COMPONENT WRAPPER [47] wraps a component and redirects invocations to that component.
- WRAPPER FACADE [39] wraps a procedural library, and redirects invocations to that library.

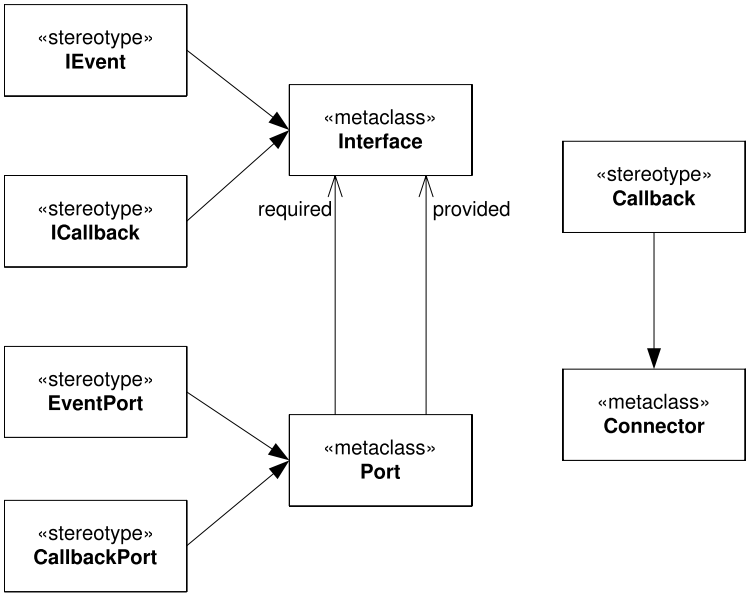

Fig. 2. Stereotypes for modeling Callback.

- A MESSAGE REDIRECTOR [45] is a component whose task it is to redirect (dispatch) invocations for a subsystem.

4.3.3. Modeling issues

The indirection structure is not explicit in structural or behavioral UML diagrams. The causes for this problem are similar to those explained for the Callback primitive because again two consecutive invocations cannot be semantically aligned. That is, the semantics are missing: is it an ordinary collaboration or an indirection?

There are similar structural and behavioral indicators as in Callback, that cause similar problems. We do not repeat these here in full detail but we provide a brief description:

- In an Indirection, clients store invocation references to proxies, which store references to targets. However, this is a vague hint because an Indirection's invocation references cannot be distinguished from ordinary references. That is, it cannot be detected from the structure that invocations are passed along the references.
- In some Indirections a standardized return type is used or one of the passed parameters is a context object for the Indirection. However, these are both occasionally used, and it is hard to detect them automatically.
- As in callback, sequence diagrams may help, but are ambiguous because again there is no semantic annotation and it is difficult to cope with temporal decoupling as well.

4.3.4. Modeling solution

To capture the semantics of indirections properly in UML and tackle the above problems, we propose the following new stereotypes and constraints:

- «*IIndirector*»: A stereotype that extends the Interface metaclass and designates the proxy component's interface to the Indirection client.
- «*ITarget*»: A stereotype that extends the Interface metaclass and designates the proxy component's interface to the Indirection target.
- «*IndirectionTargetPort*»: A stereotype that extends the Port metaclass. The «*IndirectionTargetPort*» is attached to the target component, and provides an «*ITarget*» interface, in order to accept requests from the proxy component. This can be formalized as follows:

```
-- An IndirectionTargetPort provides
-- an ITarget interface
inv: self.basePort.provided->size()=1 and
  self.basePort.provided->forAll(
    i:Core::Interface|
      ITarget.baseInterface->
        exists(j|j=i))
```

- «*IndirectionPort*»: A stereotype that extends the Port metaclass. The «*IndirectionPort*» is attached to a proxy component, requires an «*ITarget*» interface and provides an «*IIndirector*» interface. The client of the target component can connect via the «*IIndirector*» interface to the proxy component, which forwards the request to the target component through its «*ITarget*» interface. This can be expressed in OCL with the following constraints:

```
-- The IndirectionPort requires an ITarget
-- interface
inv: self.basePort.required->size()=1 and
  self.basePort.required->forAll(
    i:Core::Interface|
     ITarget.baseInterface->exists(j|j=i))


-- The IndirectionPort provides an
-- IIndirector interface
inv: self.basePort.provided->size()=1 and
  self.basePort.provided->forAll(
    i:Core::Interface|
      IIndirector.baseInterface->
        exists(j|j=i))
```

- «*Indirection*»: A stereotype that extends the Connector metaclass. It is used to connect two ports which are stereotyped as IndirectionPort and IndirectionTargetPort. The connector is constrained as follows:

```
-- An Indirection connector has only two ends
inv: self.baseConnector.end->size()=2

-- An Indirection connector connects an
-- IndirectionPort of a proxy component to a
-- matching IndirectionTargetPort of the
-- target component. An IndirectionPort
-- matches an IndirectionTargetPort if
-- the provided ITarget interface of the
-- latter matches the required
-- ITarget interface of the former.
inv: self.baseConnector.end->forAll(
  el,e2:Core::ConnectorEnd|el<>e2 implies(
    (el.role->notEmpty() and e2.role->
    notEmpty())
    and
    (if IndirectionPort.basePort->exists(p|
      p.oclAsType (Core::ConnectableElement)=
      el.role)
    then
      (IndirectionTargetPort.basePort-> exists(p|
      p.oclAsType(Core::ConnectableElement)=
      e2.role)
    and
    el.role.oclAsType (Core::Port).required=
    e2.role.oclAsType (Core::Port).provided
    and
    el.role.oclAsType (Core::Port).provided=
    e2.role.oclAsType (Core::Port).required)
  else
    IndirectionTargetPort.basePort->exists(p|
    p.oclAsType (Core::ConnectableElement)=
      el.role)
  endif)))
```

Fig. 4 illustrates these stereotypes according to the UML 2.0 Profiles package, while Fig. 5 depicts the notation used for the stereotypes.

4.4. Grouping

4.4.1. Introduction

In several design situations, a number of components belong semantically together, for instance because they fulfill a collective task. In some of these situations, designers want to model this concern explicitly using the object-oriented "part-of" (or aggregation) relationship like the ones offered by UML. In such aggregation relationships, there is one (or more) components that is (are) a "whole" which has (have) a few other components as "parts".

However, there are also other situations in which the whole is made up only from the parts, and there is no notion of a component that explicitly represents the whole. The Grouping primitive deals with such situations: A group member is part of a whole, and the whole is virtual. That is, there is no component in the software architecture for representing the group as a whole, but it is made only of its parts.

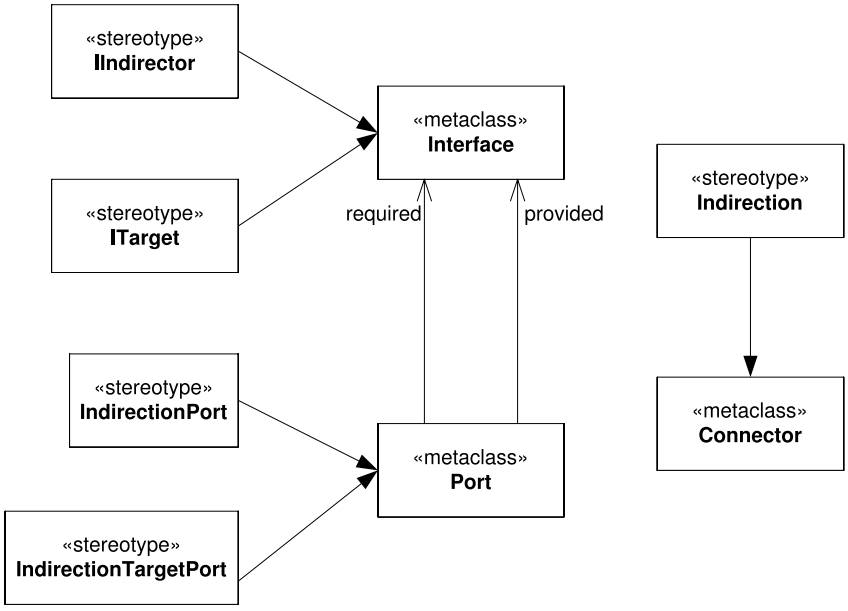

Fig. 4. Stereotypes for modeling Indirection.

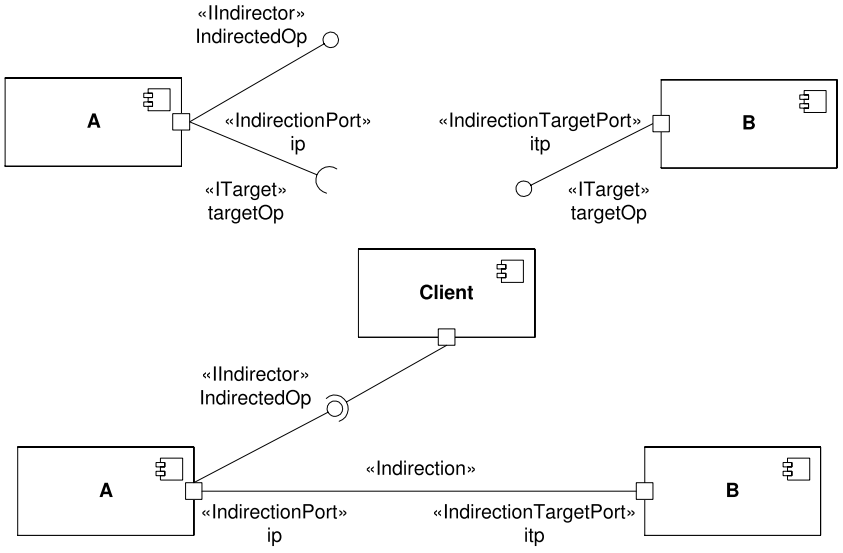

Fig. 5. The notation of the stereotypes in Indirection modeling.

4.4.2. Known uses in patterns

- The subsystem shielded by a FACADE [11] is a group of those components belonging to the subsystem.
- The LAYERS pattern [6] divides the system into logical layers, each of them is a 'virtual' group of components.
- The components of a BROKER [6] architecture are a group of components working on the same task.
- An INDIRECTION LAYER [45] redirects invocations from one group of components into another.
- A VIRTUAL MACHINE [42] redirects invocations into a group of implementation layer components.
- An INTERPRETER [42,11] redirects script invocations into a group of implementation components.
- A MESSAGE REDIRECTOR [45] dispatches invocations into a group of components that form a subsystem.
- A BLACKBOARD [6,42] is accessed and modified by a group of Knowledge Sources.
- A MICROKERNEL [6] offers services by groups of internals and external servers.
- The REFLECTION pattern [6] allows a group of application-logic components to query a group of meta-objects in order for the former to abstract their structural and behavioral aspects.
- A PEER-TO-PEER [7] system groups peer components and relates different groups with each other.
- The PUBLISH-SUBSCRIBE pattern [6,7] offers asynchronous notification to a group of independent subscribers.

4.4.3. Modeling issues

UMLs aggregation (shared aggregation) or composition (composite aggregation) relationships can be used to model part–whole relationships. According to [32] these relationships have the following semantics: Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Precise semantics of shared aggregation varies by application area and modeler. In both cases, a component is used in the relationship as an explicit whole.

But in Grouping, the whole does not really exist as a component, it is only the sum of its parts. For instance, a subsystem contains subsystem elements, but usually there is no explicit component for representing the subsystem as a whole. Hence, both UMLs aggregation or composition, model a slightly different situation, which expresses different semantics than Grouping.

Alternatively, a UML package can be used to depict such a group, but a package may own the elements, which means that a destruction of the package would also destroy the elements. On the contrary we need a more loose relationship between the group and its members.

The aggregation relationships and packages alike can contain elements other than components. Hence, it is not possible to ensure in UML that only a virtual group of components is modeled. Modelers can add other types of UML elements to the group.

4.4.4. Modeling solution

We add a simple extension to the UML meta-model for modeling groups: a stereotype «*Group*», extending the Package metaclass, is used to model a group, providing a namespace for the different group member components. We constrain the Group stereotype, so that only components can be its members, and these components are only imported and not owned by the group.

We formalize grouping in OCL using the following constraints:

```
-- A Group does not own any members
inv: self.basePackage.ownedMember->size()=0


-- All the imported members of a group are
-- Components
inv: self.basePackage.importedMember->forAll(
  oclIsTypeOf(Core::Component))
```

Fig. 6 illustrates these stereotypes according to the UML 2.0 Profiles package, while Fig. 7 depicts the notation used for the stereotypes.

4.5. Layering

4.5.1. Introduction

Layered structures are ubiquitous in software architectures, where groups of components are ordered and invocations between the different groups need to respect certain rules. For instance, the most common rule is that an intermediate layer cannot be bypassed during an invocation from a higher layer to a lower layer. As in the Grouping primitive, a layered component structure should only contain components (and not other UML elements). Also, a layer is typically a virtual entity, i.e., in many cases it only exists to indicate a conceptual abstraction in the system.

Hence, Layering builds upon the Grouping primitive and further constrains it. Specifically, it entails that group members from layer X may call into layer $X-1$ and components outside the layers, but not into layer $X-2$ and below.

4.5.2. Known uses in patterns

- The LAYERS [6] and LAYERED SYSTEM [42] patterns described layered structures.
- An OBJECT SYSTEM LAYER [47] introduces a layer hosting a object system as an extension of the language in which the OBJECT SYSTEM LAYER is implemented.
- INDIRECTION LAYER [45] describes LAYERS [6] that redirects all invocations in one system context into another.

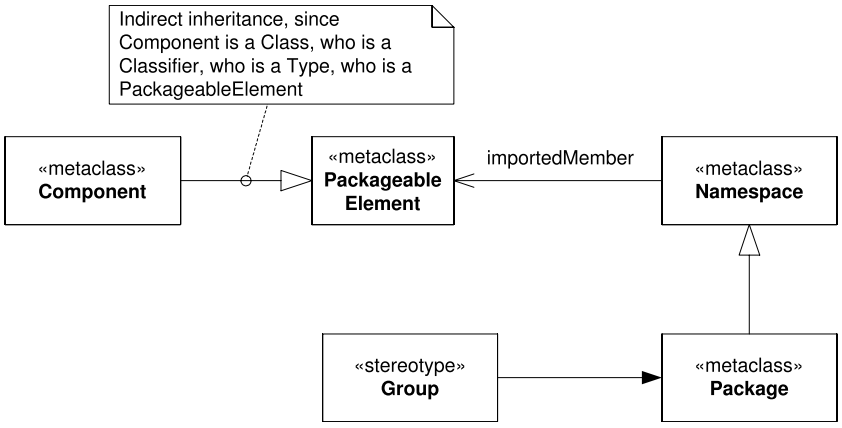

Fig. 6. Stereotypes for modeling Grouping.

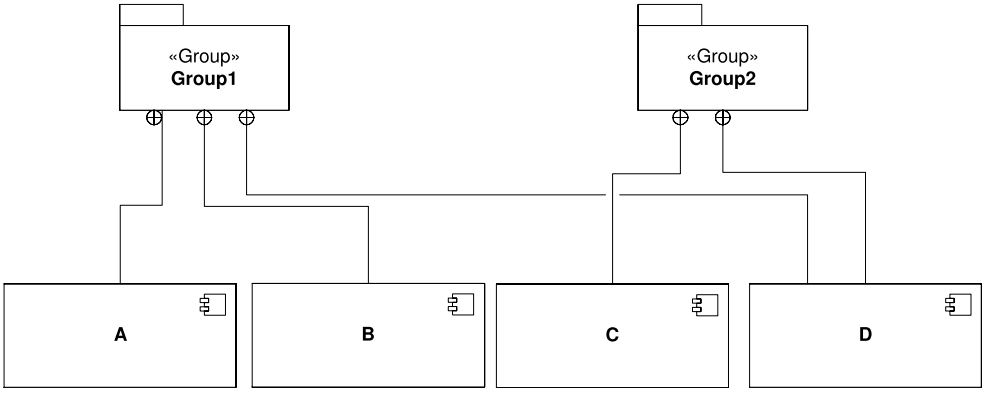

Fig. 7. The notation of the stereotypes in Grouping modeling.

- A MICROKERNEL [6] is structured in three layers: external servers, the microkernel, and internal servers.
- The PRESENTATION-ABSTRACTION-CONTROL pattern also enforces layers: a top layer with one agent, several intermediate layers with numerous agents, and one bottom layer which contains the 'leaves' agents of the tree-like hierarchy.

4.5.3. Modeling issues

The problems in modeling Layering are similar to Grouping. Hence, for the same reasons as in Grouping, the UML aggregation and composition relationships, as well as ordinary UML package structures, are not suitable to model all concerns of Layering. Additionally, we need to ensure that calls between components residing in different layers do not violate the aforementioned constraints. In contrast to groups, one layer member cannot be part of multiple layers.

4.5.4. Modeling solution

We introduce the «*Layer*» stereotype, which specializes the «*Group*» stereotype introduced above (which itself is an extension of the Package metaclass). We also impose the following constraints:

- A component can only be member of one layer and not multiple layers.
- Components who are members of layer X may call their fellow components in layer X, as well as components in layer $X-1$ but not in other layers (e.g. $X-2$ and below).

It is noted that there is no constraint about calling components in layer $X+1$ or above, since this is a specific issue to the pattern realization. Also, we introduce the tag definition layerNumber for Layers which represents the number of the layer in the ordered structure of layers. The constraints are formalized as follows:

```
-- A Layer member can only be part of one
-- layer and not multiple layers
inv: self.basePackage->forAll(p1,p2:Core::Package|
  p1<>p2 implies
    p1.importedMember->
      intersection(p2.importedMember)->isEmpty())
```

```
-- Components in Layer X may only call
-- components in the same Layer and Layer X - 1
-- but not other Layers.

inv: self->forAll(|1,|2:Layer||1<>|2 implies
```

```
if ((|l.layerNumber-|2.layerNumber).abs()>1)
then
  not |l.basePackage.ownedMember->forAll(
    c:Core::Component|
      l2.basePackage.ownedMember->
        exists(connects(c)))
else
  true
endif)
```

To realize the `connects` definition used above, the Component metaclass of UML is extended as follows:

```
-- Check whether a Component is connected
-- directly or indirectly to another component
-- through connectors
def: connects(target: Component): Boolean =
  if self.ownedPort.opposite.class->includes
  (target)
  then
    true
  else
    if self.ownedPort.opposite.class->
      exists(connects(target))
    then
      true
    else
      false
    endif
  endif
```

Fig. 8 illustrates these stereotypes according to the UML 2.0 Profiles package, while Fig. 9 depicts the notation used for the stereotypes.

4.6. Aggregation cascade

4.6.1. Introduction

A COMPOSITE [11] describes part–whole hierarchies where a composite object is composed of numerous subparts. Both composite and leaf components inherit from the same class, and are treated uniformly by clients. For example a GUI widget can call its parts to paint themselves, and they call their parts and so on. A cascade [9] is a COMPOSITE structure with (recursive) constraints of the form: "A composite *A* can only aggregate components of type *B*, *B* only *C*, etc".

Such interconnected or recursive COMPOSITE structures of components are in fact a common concern in object-oriented systems. An Aggregation Cascade models this situation, but it does not define the precise semantics of the aggregation relationships between the COMPOSITE structures. Instead, these are to be defined by the application domain and the architect.

4.6.2. Known uses in patterns

- The COMPOSITE [11] pattern describes general composite structures. Our primitive concerns especially design situations with multiple composite structures that are interconnected or recursive (and may have additional constraints).
- A CASCADE [9] is a COMPOSITE structure with (recursive) constraints of the form: "A composite *A* can only aggregate components of type *B*, *B* only *C*, etc".
- ORGANIZATION HIERARCHY [10] is an analysis pattern that requires both composite constraints and (recursive) constraints of the form: "A composite *A* can only aggregate components of type *B*, *B* only *C*, etc". Such analysis patterns are frequently realized by component architectures.

4.6.3. Modeling issues

For this primitive, we could consider the UML Aggregation, which is a special form of the UML Association. Because it depicts a part/whole relationship, but the precise semantics of shared aggregation varies by application area and modeler (see [32]), it is the UML modeling element that matches the Aggregation Cascade primitive concerns the closest.

Through Aggregation, a whole aggregates parts, and a part cannot contain its whole, but it is possible for a part to be aggregated in multiple wholes. That is, links between

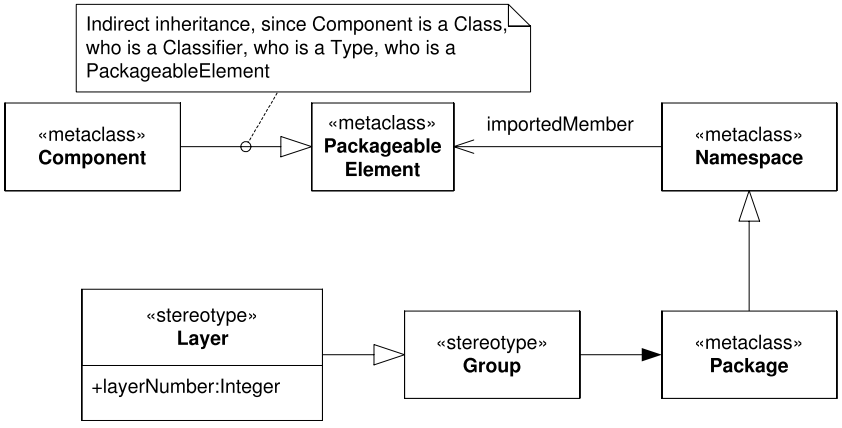

Fig. 8. Stereotypes for modeling Layering.

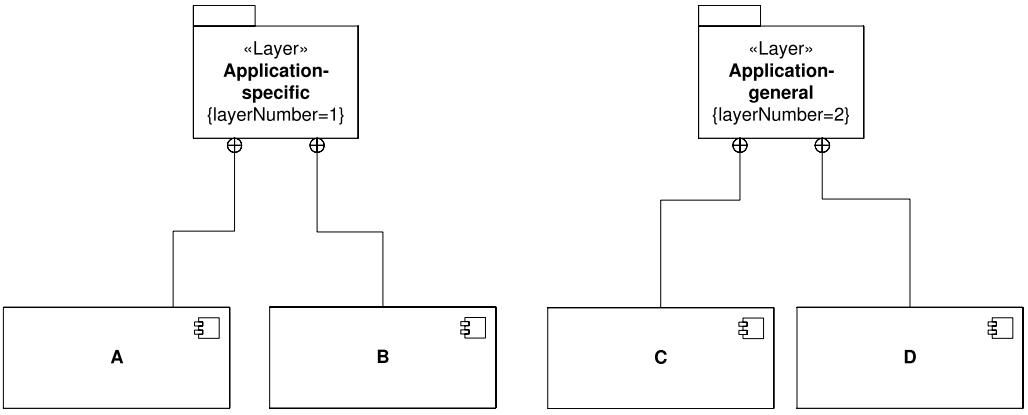

Fig. 9. The notation of the stereotypes in Layering modeling.

hierarchies are possible, but not circular links. In our primitive, the composites call their parts recursively, and there are recursive composition constraints. UMLs aggregation, however, cannot perform such recursive calls or 'cascading' constraints.

4.6.4. Modeling solution

We constrain all components of the hierarchy, composites and leafs, to inherit from the same component type. Furthermore we define a stereotype «*AggregationCascade*» as an extension of the stereotype «*Indirection*», which itself extends the Connector metaclass. An Aggregation Cascade connects a composite to its parts. It extends Indirection since it forwards the recursive operations to clients. Since it specializes Indirection all the constraints from Indirection are also valid here.

The Association that types the Connector is an Aggregation, to enforce that this is really a connector between a composite and its parts. Since we introduce the aggregation between two specific, connected components, and not between a Composite and a generic interface (as in the COMPOSITE pattern), these aggregations are constrained so that "A composite *A* can only aggregate components of type *B*, *B* only *C*, etc".

These constraints can be formalized as follows:

```
-- There is always an association that types
-- the AggregationCascade and that association
-- is an Aggregation. Note that the association
-- being an aggregation implies that it is also
-- binary (only binary associations can be
-- aggregations)
inv: self.baseConnector.type->size()=1 and
    self.baseConnector.type.memberEnd->
      exists(aggregation=
        Core::AggregationKind::shared)

-- The association is navigable both ways
-- (so the classes own the association ends)
inv: self.baseConnector.type.ownedEnd->isEmpty()

-- Component A can only aggregate components
-- of the same type B
inv: let componentA:Core::Class =
  self.baseConnector.type.memberEnd->
    select(aggregation=
        Core::AggregationKind::shared).
        class->any(true) in
      componentA.ownedAttribute.opposite.
      class->forAll(cl,c2:Core::Class|
          cl<>c2 implies cl.name=c2.name)
-- All components of the hierarchy inherit
-- from the same type
inv: self.baseConnector.type.memberEnd.class->
  forAll(cl,c2:Core::Class|cl<>c2 implies
    cl.parents()->intersection
    (c2.parents())->
      notEmpty())
```

Fig. 10 illustrates these stereotypes according to the UML 2.0 Profiles package, while Fig. 11 depicts the notation used for the stereotype using an example: a model according to the ORGANIZATION HIERARCHY [10] analysis pattern.

4.7. Composition Cascade

4.7.1. Introduction

A Composition Cascade builds upon Aggregation Cascade, and further enforces that a component may not be part of more than one composite at any time. In this case, composites have a lifecycle responsibility for their parts. That is, the whole may take direct responsibility for creating or destroying the parts, or it may accept an already existing part, and later pass it on to some other whole that assumes responsibility for it.

Again, these lifecycle operations need to be applied in a recursive fashion: e.g. a composite that is destroyed, destroys its parts, which recursively destroy their parts, and so on.

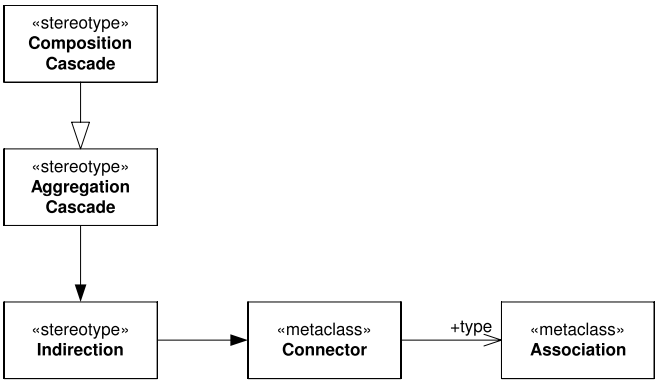

Fig. 10. Stereotypes for modeling Aggregation and Composition Cascades.

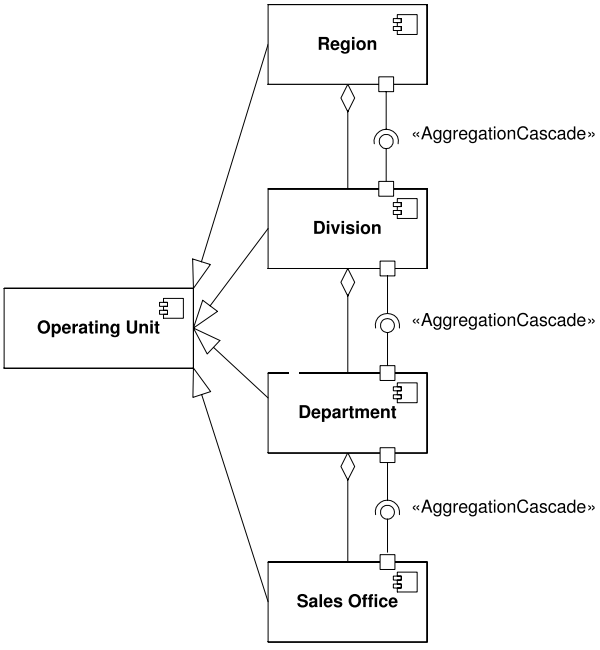

Fig. 11. Example of an aggregation cascade.

4.7.2. Known uses in patterns

Composition Cascade has the same known uses in patterns as Aggregation Cascade: COMPOSITE [11], CASCADE [9], and ORGANIZATION HIERARCHY [10]. The difference to the Aggregation Cascade known uses is that the patterns are realized using aggregation relationships that assume lifecycle responsibility for the parts.

4.7.3. Modeling issues

We face the same modeling issues as in Aggregation Cascade, but we need to model a more rigid aggregation relationship: A component may not be part of more than one composite at any time. The recursive operations must also include the aforementioned lifecycle operations.

4.7.4. Modeling solution

The modeling solution is to extend the Aggregation Cascade primitive and add additional constraints on the «*Aggregation Cascade*» Connector. We thus define the «*CompositionCascade*» stereotype as an specialization of «*AggregationCascade*». In this case the Association that types the connector is a Composite Aggregation, so each part can only be owned by one Composite.

We thus only have to add one more constraint. Of course, the rest of the constraints from AggregationCascade hold here.

```
-- The association that types the
-- CompositionCascade is a CompositeAggregation
inv: self.baseConnector.type.memberEnd->exists(
  aggregation=Core:: AggregationKind::composite)
```

Fig. 10 illustrates these stereotypes according to the UML 2.0 Profiles package, while Fig. 12 depicts the notation used for the stereotype.

4.8. Shield

4.8.1. Introduction

In certain cases, a set of components cannot or should not be accessed directly by clients. Instead, another intermediary component is to be used to access the set of components. The rationale behind this 'shielding' is usually information hiding, separation of concerns, or the implementation of central tasks which should be respected by all the components in the set.

The Shield primitive captures this design rationale with the following properties: One or more components act as 'shields' for a set of components that form a subsystem. No external client should be allowed to access members of the subsystem directly, but access should happen only through these 'shields'.

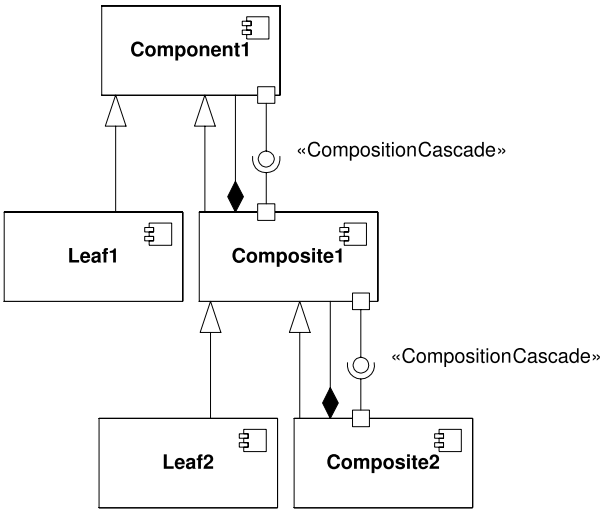

Fig. 12. Notation of a composition cascade.

4.8.2. Known uses in patterns

- The subsystem shielded by a FACADE [11] often should not be accessed directly, but only via the FACADE.
- In the LAYERS pattern [6], often layers should not get bypassed, i.e., lower-level layers should not be accessed directly. Also, often layer elements should only be accessed via the layer's interface. Hence, the layer's interface shields the layer's elements and other lower-level layers.
- The INDIRECTION LAYER [45] shields the "target" component from the client.
- A MESSAGE REDIRECTOR [45] is a component whose task it is to redirect (dispatch) invocations for a subsystem. It should usually not get bypassed.
- In the REFLECTION pattern [6], the meta-object protocol shields the meta-objects from the client component.
- A VIRTUAL MACHINE [12] shields the platform details in order for the byte-code to be ported in different platforms.
- An OBJECT SYSTEM LAYER [47] introduces a layer hosting a object system as an extension of the language in which the OBJECT SYSTEM LAYER is implemented. The objects in the OBJECT SYSTEM LAYER should only be accessed through the OBJECT SYSTEM LAYER'S interface.
- Many remoting patterns [44] used in a layered BROKER architecture shield a whole subsystem realizing their functionality: The subsystem's should not be accessed directly.

4.8.3. Modeling issues

We need to model the members of the subsystem, as well as the components shielding the subsystem. Here, the problems in modeling the Shield primitive are similar to Grouping. Hence, for the same reasons as in Grouping, the UML aggregation and composition relationships, as well as ordinary UML package structures, are not suitable to model subsystems.

Additionally, we need to make sure that no invocation can bypass the 'shield' components. This concept also cannot be represented in standard UML. For instance, if we model the subsystem as a Group following the Grouping primitive, any element of the Group's Package can be accessed from outside and is visible to clients. The imported package member that are used to model Grouping offer no means to limit the access to a Group member.

4.8.4. Modeling solution

We utilize the Grouping primitive (or extensions of it such as Layering), described above to model the membership of the components in the 'shielded' group.

We introduce the stereotype «*IShield*» that extends the Interface metaclass. «*IShield*» is offered by the components that shield the subsystem and provide access to the rest of the group members. We use UMLs Visibility Kind abstraction to make an IShield interface a public interface, and add the constraint that all IShield interfaces must be group members. This can be formalized in OCL as follows:

```
-- The visibility of the methods of IShield are
-- declared public so that any client can access
-- it directly
inv: self.baseInterface.feature->forAll(f |
  f.visibility = Core::VisibilityKind::public)


-- IShield interfaces are provided by a member
-- of a group
inv: self.baseInterface->forAll(i|
  Core::Package.importedMember.oclAsType
  (Core::Component).
    provided->includes(i))
```

We also introduce the stereotype «*Shield*» that extends the Connector metaclass. A «*Shield*» connector can be used by a client to connect to the "shield" component. Thus we constrain «*Shield*» to match the provided «*IShield*» interface of a "shield" component to the matching required interface of a client component. «*Shield*» is constrained as follows:

```
-- A Shield Connector has only two ends
inv: self.baseConnector.end->size()=2


-- There is always an association that types
-- the Shield and that association is navigable
-- both ways so the classes own the association
-- ends (preconditions so that
-- Property. opposite is not empty)
inv: self.baseConnector.type->size()=1


inv: self.baseConnector.type.ownedEnd->isEmpty()


-- A Shield Connector matches the provided
-- IShield interface of a shield component
-- to the matching required interface of a
-- client component.
inv: self.baseConnector.end->forAll(
  el,e2:Core::ConnectorEnd|el<>e2 implies (
    (el.role->notEmpty() and e2.role->notEmpty())
    and
    ((el.role.oclAsType (Core::Port).required=
      e2.role.oclAsType (Core::Port).provided)
    and
    (el.role.oclAsType(Core::Port).required->
    forAll(i|IShield.baseInterface->
      exists(j|j=i))))
    or
      ((el.role.oclAsType (Core::Port).provided=
        e2.role.oclAsType (Core::Port).required)
      and
```

```
    el.role.oclAsType(Core::Port).provided->
    forAll(i|IShield.baseInterface->
      exists(j|j=i)))))
```

Finally, we introduce the stereotype «*ShieldPort*» that extends the Port metaclass. A port stereotyped as «*ShieldPort*» provides at least one «*IShield*» interface. «*ShieldPort*» is also extended by a tag definition, shieldGroup, for denoting the group which is shielded. «*ShieldPort*» is constrained so that all components that connect to its port and are not client components, should be members of the shieldGroup. Finally, each such component that is not itself a shield component for the same or other groups, should have a "package" visibility for all its provided interfaces. That means, the member components of the shielded group can only be accessed by other members of the group or via the «*IShield*» interfaces. These constraints of «*ShieldPort*» are formalized in OCL as follows:

```
--A shield port provides one or more interfaces,
--and one of them is an IShield interface
inv: self.basePort.provided->size()>=1 and
  self.basePort.provided->forAll(
    i:Core::Interface|IShield.baseInterface->
    exists(j|j=i))


--All components connected to this port who are
--not client components (require the same
--IShield that self provides) are members of the
--same group and that group has the same name
--as the tagged value "shieldGroup"
inv: let
ShieldedComponents:Bag(Core::Component) =
  self.basePort.opposite->reject(p:Core::Port|
    p.required->includes(self.basePort.provided)).
      class.oclAsType(Core::Component) in
    Groupings::Group.basePackage->
    one(importedMember->
          includesAll(ShieldedComponents) and
          name = self.shieldGroup)
    and
--for each such component c, who does not
--provide an IShield interface, all provided
--interfaces of c are of visibility "package"
    ShieldedComponents.ownedPort->
      reject(p:Core::Port|
      Shields::IShield.baseInterface->
        includesAll(p.provided))->forAll
      (p:Core::Port|p.provided.feature->
      forAll(f|f.visibility=
        Core::VisibilityKind::package))
```

Fig. 13 illustrates these stereotypes according to the UML 2.0 Profiles package, while Figs. 14 and 15 depict the notation used for the stereotypes.

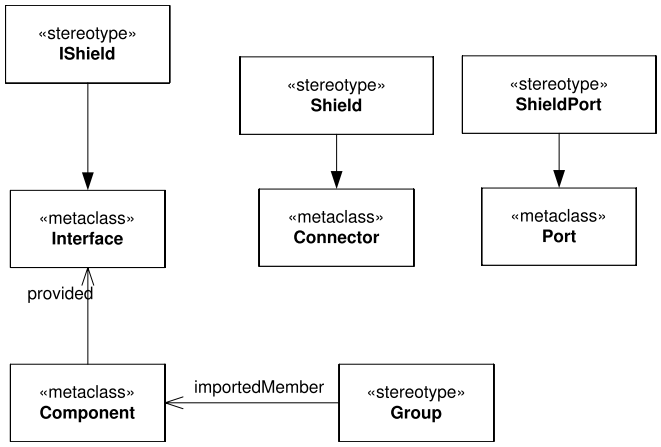

Fig. 13. Stereotypes for modeling Shields.

4.9. Typing

4.9.1. Introduction

In many situations, the typing abstraction provided by the design or programming language is not sufficient for modeling domain types. For instance, the domain might require dynamic or constrained type dependencies.

Consider for example a typical business situation: There are different Party Types in a company (e.g. "manager", "implementation group"), and a particular business entity (e.g. John, group X) can change its Party Type at runtime: A component of party type "manager" can become "senior manager", a group of type "test group" can become "implementation group", and so forth. There are usually constraints on these type changes (e.g. a group cannot take a Party Type that needs to be fulfilled by a person).

The only abstraction that can be used in these cases, is the generic association, but that does not include the semantics of dynamic or constrained typing. A custom, dynamic type system for Party Types needs to be implemented from scratch by the developers. The Typing primitive introduces the notions of a supertype connector and a type connector, which can be used to define custom typing models using associations.

4.9.2. Known uses in patterns

- The pattern TYPE OBJECT [21] resolves the problem that a certain type relationship has to be dynamic in a statically typed, object-oriented language. By building the type relationship with the objects of the language, instead of the static classes, dynamic typing is "simulated" using delegation.
- A common example of an extension of the TYPE OBJECTS [21] pattern are analysis pattern that realize a KNOWLEDGE LEVEL [10], a meta-level architecture for typing in the sense of TYPE OBJECT. We give below the examples of party types and accountability types.

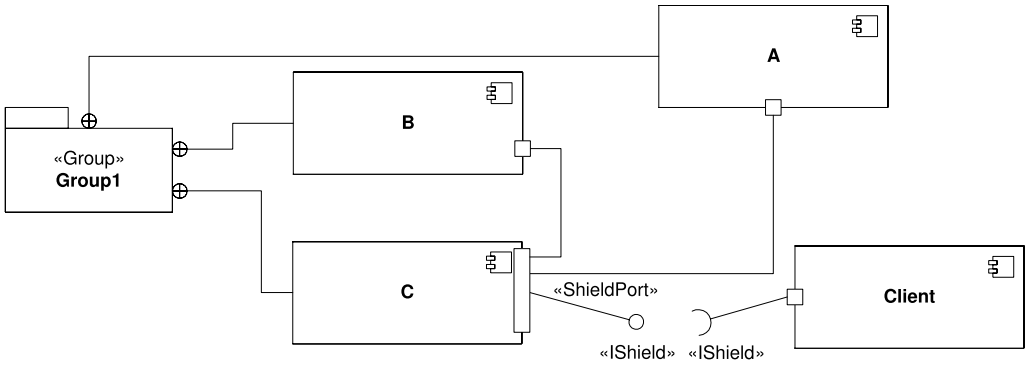

Fig. 14. The notation of the stereotypes in Shield modeling (1).

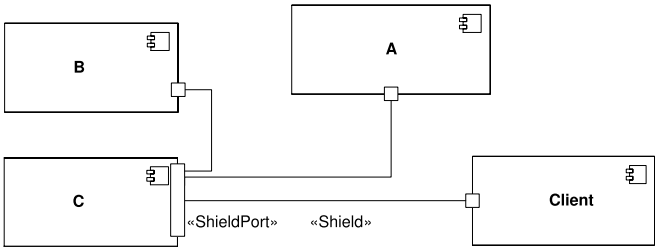

Fig. 15. The notation of the stereotypes in Shield modeling (2).

- An OBJECT SYSTEM LAYER [47] introduces a layer hosting a object system as an extension of the language in which the OBJECT SYSTEM LAYER is implemented. Thus a whole custom type system is introduced with in the OBJECT SYSTEM LAYER.

4.9.3. Modeling issues

In UML2 the Generalization metaclass is used to model inheritance. A generalization is a relationship between a more general classifier and a more specific classifier. The specific classifier inherits the features of the more general classifier. The InstanceSpecification metaclass is used to define a model element that represents an instance of a Classifier. Those metaclasses exactly match our concern to model a type system, and of course they can be extended to model custom aspects of it. However, in UML there is no notion of changing a model derived from the UML2 meta-model at runtime. For instance, a reclassification of an instance, or a change of the supertype, are not supported by UMLs Generalization and InstanceSpecification metaclasses.

UML supports associations as relationships that are changeable at runtime. However, associations are only changeable at the instance level. Typing, in contrast, requires to express a relationship between Classifiers, or Classifiers and their instances.

Additionally, the model cannot make explicit the fact that a typing relation is modeled, because the relationship looks like an ordinary association. The semantics of typing, such as type compliance rules, type conversion rules, inheritance, etc., are only implicit and not documented. Constraints of the typing relation are also not documented as such.

4.9.4. Modeling solution

We introduce components that represent types at runtime. These components for types form a meta-level or type-level. Between any ordinary component and a type-level component, a Connector can be stereotyped as being a Type Connector. This Connector depicts an instance of relationship. Between two elements of the type-level, a Connector can be stereotyped as being a Super Type Connector. This Connector depicts an inheritance relationship.

We introduce two stereotypes that extend the Connector metaclass, and realize these typing relationships:

- «*TypeConnector*» realizes the typing relationship (using the associated «*TypeConnectorBehavior*»). It has a constraint to avoid circular type dependencies:

```
-- A Type Connector has only two ends
inv: self.baseConnector.end->size()=2


-- A Type Connector might not be applied
-- in circular order
inv: self.baseConnector.end.role->forAll(
  cl,c2:Core::ConnectableElement|cl<>c2 and
  cl.oclAsType(Core::Port).class =
  c2.oclAsType(Core::Port).class implies
    not cl.typeConnection(c2))
```

- «*SupertypeConnector*» realizes the supertype relationship (using the associated «*SupertypeConnectorBehavior*»). It has a constraint to avoid circular supertype dependencies:

```
-- A Super Type Connector has only two ends
inv: self.baseConnector.end->size()=2


-- A Super Type Connector might not be
-- applied in circular order
inv: self.baseConnector.end.role->forAll(
  cl,c2:Core::ConnectableElement|cl<>c2
  and
  cl.oclAsType(Core::Port).class =
```

```
    c2.oclAsType(Core::Port).class implies
      not cl.supertypeConnection(c2))
```

The two constraints above check for direct and indirect circularity of the type relationships using the `typeConnection` and `supertypeConnection` definition, which are defined for the UML metaclass Component:

```
def: typeConnects(target: Component):Boolean =
  if self.ownedPort.opposite.class->
  includes (target)
    and Typings:: TypeConnector.baseConnector.
    end.role. oclAsType(Property).class->
      includesAll(Set{self,target})
  then
    true
  else
    if self.ownedPort.opposite.class->
       exists(connects(target))
    then
      true
    else
      false
    endif
endif
def: superTypeConnects(target: Component):Boolean =
  if self.ownedPort.opposite.class->
  includes (target)
    and Typings::SuperTypeConnector. baseConnector.
    end.role.oclAsType(Property).class->
    includesAll(Set{self,target})
  then
    true
  else
    if self.ownedPort.opposite.class->
      exists(connects(target))
    then
      true
    else
      false
    endif
endif
```

Using these Connectors we can model a custom-built type system. For instance, in the example above we can make the Party component have a «*TypeConnector*» to a specific Party Type "manager", which itself has a «*SupertypeConnector*» to a generic "party type" class. Using this meta-model, we can derive instances, representing different parties and party types, and we can provide the respective constraints both on the instance-level and the meta-level.

Fig. 16 illustrates these stereotypes according to the UML 2.0 Profiles package, while Fig. 17 depicts the notation used for the stereotypes.

Fig. 18 illustrates an example of how a typing meta-model can be built according to the Party Type example given before. We introduce an additional Accountability Type. The component instances derived from this model realize typed components (Party, Accountability, and specialization of these), and meta-descriptions for these types (Party Type, Accountability Type, and specialization of these). Both, Party and Accountability instances can dynamically change their types because types are realized as runtime components. Also we can provide constraints between these types, such as the ones depicted in the figure. Thus the result is a dynamic and constrained type system.

4.10. Virtual Connector

4.10.1. Introduction

In many patterns and larger architectures, components have no direct relationship, but still communicate virtually using other components and connectors in between. For instance, in a layered distributed client/server architecture a component on the client-side often virtually communi-

Fig. 16. Stereotypes for modeling Typing.

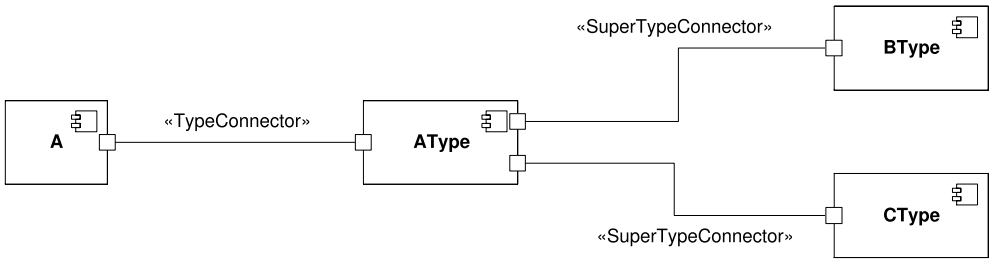

Fig. 17. The notation of the stereotypes in Typing modeling.

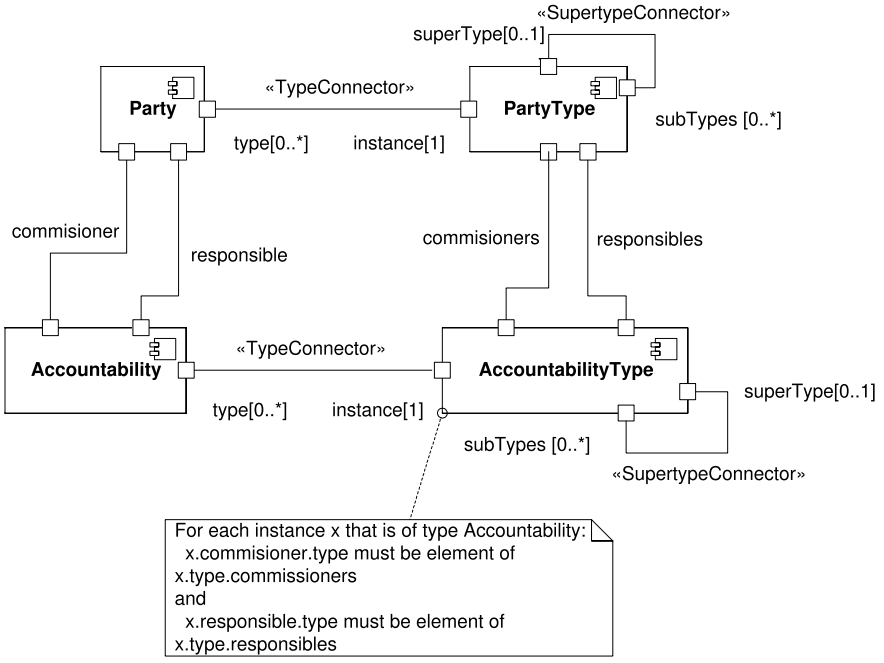

Fig. 18. Example of typing: Party Type and Accountability Type.

cates with a component from the same layer on the server-side.

The Virtual Connector primitive models this concern.

4.10.2. Known uses in patterns

- The client-side components and the server-side components of a BROKER [6] communicate virtually among each other.
- Many remoting patterns [44] virtually communicate with each other, for instance: client and server INVOCATION INTERCEPTORS, REQUESTOR and INVOKER, client and server MARSHALLERS, CLIENT and SERVER REQUEST HANDLER, and client and server PROTOCOL PLUG-INS.
- PROXIES [11] often use intermediate components and thus virtually communicate with their target. For instance, remote PROXY [6] use a BROKER [6] to access the remote target.

4.10.3. Modeling issues

The virtual relationship is an important additional information, but is not explicit in a UML diagram. It must be deduced from the implicit collaboration of components and connectors. If multiple virtual dependencies exist in the same architecture, as for instance in distributed layers, it cannot be deduced which component corresponds with which other component without further documentation.

In standard UML the virtual relationship can only be modeled by introducing another explicit connector or association between the component. Then, however, we cannot distinguish the virtual communication from non-virtual communication in the models anymore.

4.10.4. Modeling solution

We introduce a stereotype «*VirtualConnector*» as an extension of the Connector metaclass. This connector is used between components that have a virtual relationship. We further define the stereotype «*IVirtual*», as an extension of the Interface metaclass. Therefore a «*VirtualConnector*» matches an «*IVirtual*» Interface of one component to another. We enforce the constraint that the «*VirtualConnector*» can only be used between two components *A* and *B*, if there is a path of components and connectors that link *A* to *B*. For instance, if *A* is connected to *C*, *C* is connected to *D*, and *D* is connected to *B*, then a «*VirtualConnector*» from *A* to *B* might be introduced.

We can formalize the constraints as follows:

```
-- A Virtual Connector has only two ends
inv: self.baseConnector.end->size()=2


-- A Virtual Connector matches the provided
-- IVirtual interface of one component to
-- to the matching required interface of another.
inv: self.baseConnector.end->forAll(
  el,e2:Core::ConnectorEnd|el<>e2 implies (
  (el.role->notEmpty() and e2.role->notEmpty())
  and
  ((el.role.oclAsType (Core::Port).required=
    e2.role.oclAsType (Core::Port).provided)
  and
 (el.role.oclAsType(Core::Port).required->
 forAll(i|IVirtual.baseInterface->
    exists(j|j=i))))
```

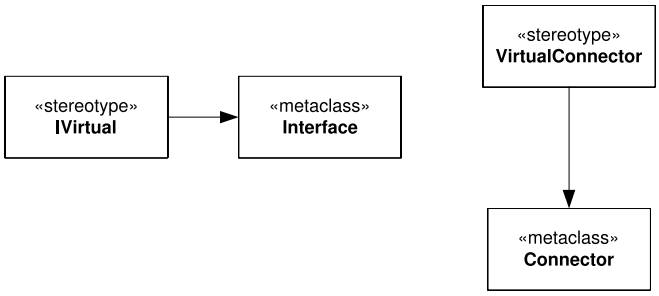

Fig. 19. Stereotypes for modeling Virtual Connector.

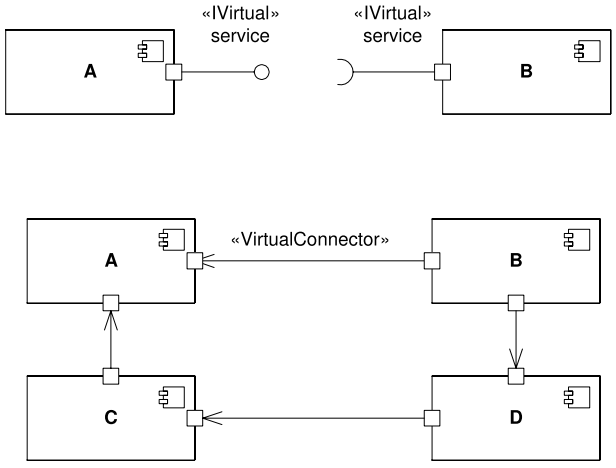

Fig. 20. The notation of the stereotypes in Virtual Connector modeling.

```
    or
 ((el.role.oclAsType (Core::Port).provided=
    e2.role.oclAsType (Core::Port).required)
    and
    el.role.oclAsType(Core::Port).provided->
    forAll(i|
      IVirtual.baseInterface->
      exists(j|j=i)))))


-- A VirtualConnector can only be used between
-- two components A and B, if there is a path of
-- components and connectors that link A to B.
inv: self.baseConnector.end.role.
  oclAsType(Core::Property).class->forAll(
    c1,c2:Core::Component|c1<>c2 implies
      c1.oclAsType
      (Core::Component).connects(c2))
```

Fig. 19 illustrates these stereotypes according to the UML 2.0 Profiles package, while Fig. 20 depicts the notation used for the stereotypes.

5. Case study

Leela [46] is an infrastructure that provides a federated model of remote peers, thus offering loosely coupled services. Within a federation, all peers are equal, they can offer Web services (and possibly other kinds of services) to other peers, and they can connect spontaneously to other peers (and to the federation). Each remote object can potentially be part of more than one federation as a peer, and each peer decides which services it provides to which federation. Certain peers in a federation may be able to access extra services that are not offered to other peers in this federation, via their partaking in other federations. Leela peers are hosted by Leela applications. One such application can host multiple peers and federations.

Leela is implemented using the architectural patterns from [44]. In our first attempt to design the system, we used the standard UML class diagrams [46]. However, the architectural patterns could not be explicitly modeled and therefore the design decisions taken that were concerned with these patterns are not documented, except as complementary meta-information to the class diagrams. This meta-information can be textual or it can make use of a formal notation, nevertheless it is not part of the UML diagrams. To overcome this problem, we have applied our UML profile to explicitly model the architectural components, connectors, configurations, and constraints in Leela's design. Due to space constraints, as a case study we present an excerpt of this design: the basic communication framework of Leela.

5.1. Broker architecture

Leela implements a BROKER [6], which suggests a general architectural configuration that separates a distributed system's communication functionality from its application functionality by isolating all communication-related concerns. A BROKER hides and mediates all communications between the objects or components in a system. Local client-side and server-side brokers enable the exchange of requests and responses between the peers.

Each peer in Leela acts as a client and a server at the same time. Thus, Leela peers are composite components that contain both client-side and server-side BROKER subcomponents. In the following description, the BROKER is viewed as a compound pattern that is implemented using several patterns from the Remoting pattern language [44]. Even though client-side and server-side BROKER components are present in the same system, it makes sense to distinguish client-side and server-side roles of the components in order to make the pattern-based architecture more understandable. Unfortunately, this cannot be easily modeled with UML because the BROKER as a whole is not an explicit component, but consists of several components. Thus we cannot use UML composition or aggregation relationships here. However, the Grouping primitive from our UML profile is an ideal match. We introduce two «*Group*» packages: ClientBroker and ServerBroker. For each BROKER component, we add a namespace relationship either to ClientBroker package or ServerBroker package, indicating membership to the respective group. The group mem-

bership of the components introduced, is depicted in Fig. 21.

5.2. Basic invocation architecture

Fig. 22 shows the basic software architecture diagram of Leela, using our profile. A BROKER consists of a client-side REQUESTOR [44] to construct and forward invocations, and a server-side INVOKER [44] that calls the target peer's operations. A MARSHALLER [44] on each side of the communications path handles the transformation of requests and responses from programming-language-native data types into byte arrays that can be sent over the wire.

As its basic communication resource each Leela application uses a component, called the `RequestHandler`, that implements both a CLIENT REQUEST HANDLER [44] and a SERVER REQUEST HANDLER [44]. The CLIENT REQUEST HANDLER forwards request messages from a client to the server. The SERVER REQUEST HANDLER receives these requests at the server-side, and triggers the invocation of the peer. Because `RequestHandler` realizes both patterns, it is member of both groups, ClientBroker and ServerBroker.

The request handlers contain PROTOCOL PLUG-INS [44] for the various protocols that transport the message across the network. Currently, Leela supports PROTOCOL PLUG-INS [44] for various SOAP implementations. However, virtually any other communication protocol can be used as well, because Leela's MARSHALLER [44] uses a simple string-based format as a message payload, and (re-)uses Tcl's automatic type converter to convert the string representations to native types and vice versa.

There are a number of further design issues which need to modeled. First of all, the application of the Remoting patterns leads to an architecture based on the LAYERS pattern [6]. The same layers are present on client and server-side: Protocol, RequestHandling, Invocation, and Application. We model the layers according to our Layering primitive. For each layer, we introduce a «*Layer*» package and the tagged value receives the respective layer number. Each layered component is imported to the corresponding layer. Fig. 23 shows the layer membership of the components discussed in this section. There a number of constraints:

- Components from the layer Application can only interact with components from the layers Application and Invocation, or components who are not part of a layer.
- Components in the layer Invocation can only be accessed via `Invoker` or `Requestor`, through a Shield Connector. That is, all internal interfaces are stereotyped «*IShield*».
- Components from the layer Invocation can only interact with components from the layers Invocation and RequestHandling, or components who are not part of a layer.
- Components in the layer RequestHandling can only be accessed via the `RequestHandler` component through a Shield Connector. That is, all internal interfaces are stereotyped «*IShield*».
- Components from the layer RequestHandling can only interact with components from the layers RequestHandling and Protocol, or components who are not part of a layer.
- Components from the layer Protocol can only interact with components from the layer Protocol or components who are not part of a layer.

Note that these constraints apply for client-side and server-side components. The client-side and server-side components are distinguished using the Grouping primitive.

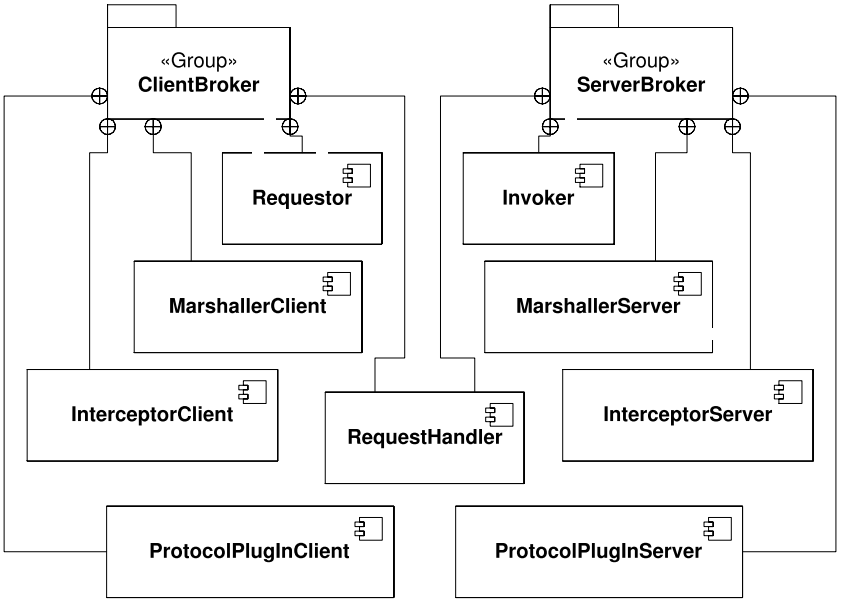

Fig. 21. Group membership of the Leela components.

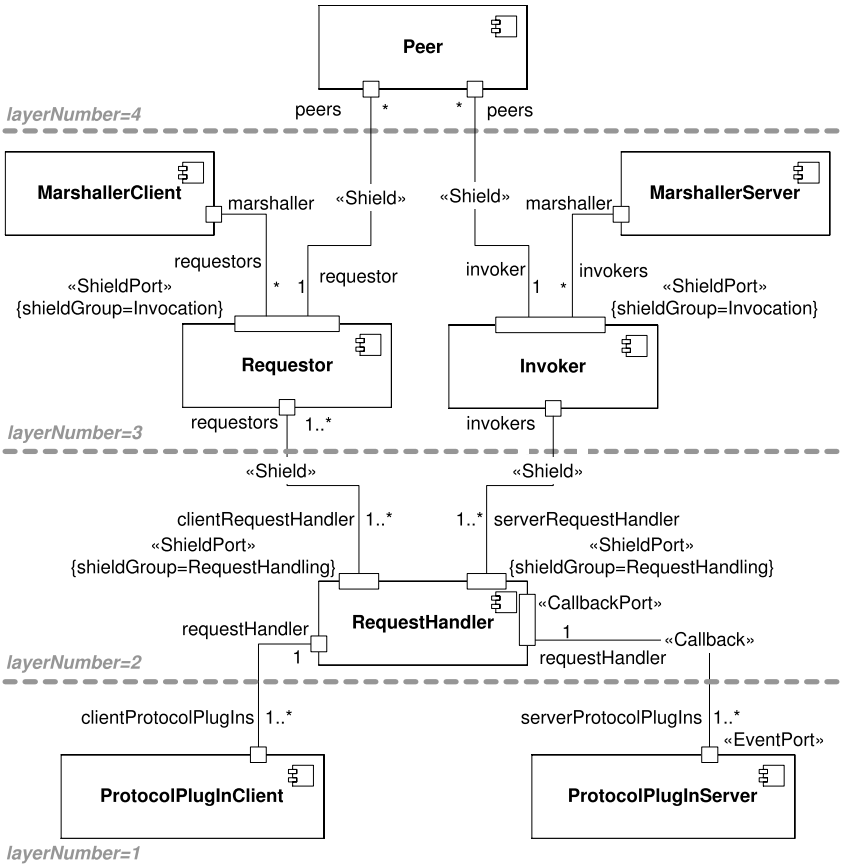

Fig. 22. Basic, broker-based invocation architecture.

The client-side PROTOCOL PLUG-IN is simply invoked by the request handler component. The server-side PROTOCOL PLUG-IN, however, receives requests and result messages from the network asynchronously (it contains a REACTOR [39] implementation). Thus the request handler is informed of network events using callback events. This is modeled using our Callback primitive (see Fig. 22).

In addition a virtual communication between the respective components at each layer of the BROKER architecture happens. This is modeled using the Virtual Connector primitive, as shown in Fig. 24.

So far we have only modeled the base components. In the next sections, let us take a closer look at two exemplary component types: peers and interceptors.

5.3. Peers and federations

As aforementioned, two different kinds of peers exist: ordinary peers and federations of peers. Federations of course contain peers, but this cannot be properly modeled with UMLs composition or aggregation relationship alone because we require a constrained relationship here. Thus we model federations as special, composite peers that are connected through an Aggregation Cascade to other peers with the following constraints:

- A peer can be part of multiple federations. That's why we use Aggregation Cascade and not Composition Cascade.
- A federation cannot contain peers of the type federation, unless they are federation proxies (see below).
- A federation proxy (see below) cannot contain other peers.

Peers can interact with other peers using the REQUESTOR, which realizes the virtual communication link. Sometimes it is more handy to use the pattern CLIENT PROXY [44]: A CLIENT PROXY is a placeholder for the peer in the client process. By presenting clients with an interface that is the same as the peer's, the proxy lets the client interact with the peer as if it were a local object. Internally, the CLIENT PROXY transforms the invocations it receives into REQUESTOR invocations. Leela also supports peer and federation proxies that act as CLIENT PROXIES, offering the interfaces of a peer or federation. The proxies thus provide indirections, which can be modeled using the Indirection primitive. We have realized the proxies slightly different from the PROXY pattern in [11]. In order to have "real" proxies we need two more constraints for the indirections in Fig. 25:

- A peer proxy cannot have peers of the types peer proxy or federation as indirection targets.

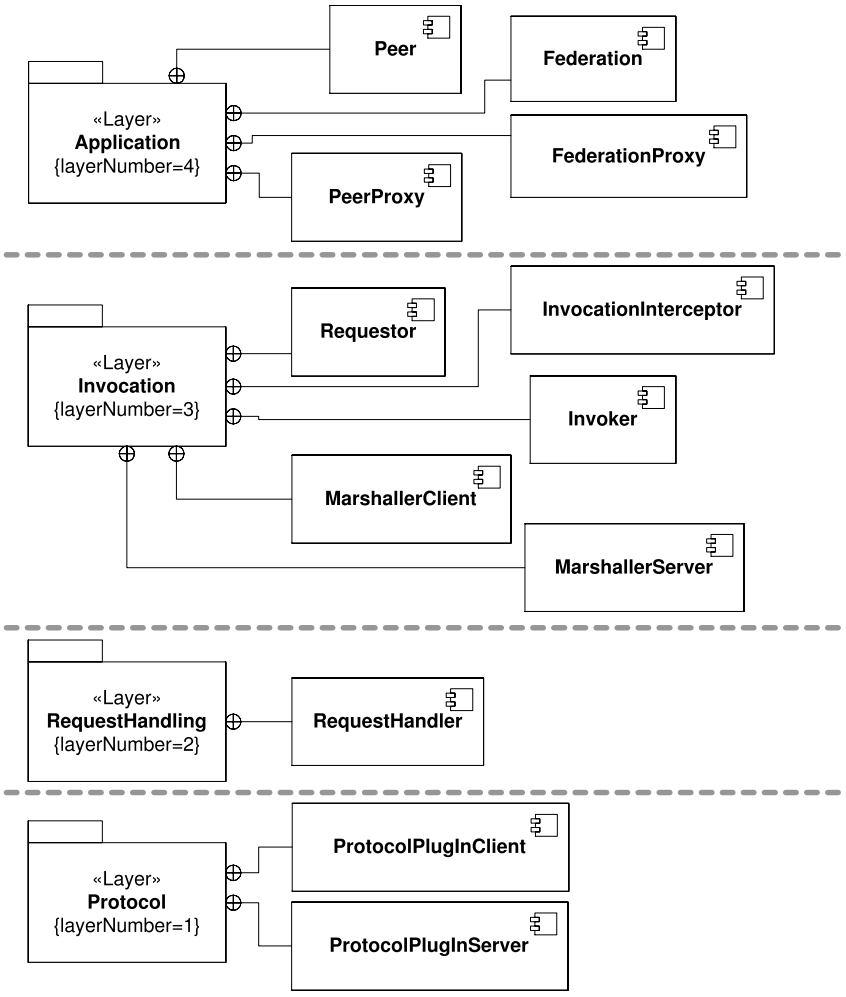

Fig. 23. Layers of the Leela architecture.

- A peer proxy cannot have peers of the types federation proxy as indirection targets.

5.4. Invocation interceptors

The Leela invocation chain on the client-side and the server-side is based on INVOCATION INTERCEPTORS [44], which transparently extend the invocation on both sides with new behavior. The most prominent task of the INVOCATION INTERCEPTORS in Leela is control of remote federation access. On the client-side, an INVOCATION INTERCEPTOR intercepts the construction of the remote invocation and adds all federation information for a peer into the INVOCATION CONTEXT [44]. On the server-side this information is read by another INVOCATION INTERCEPTOR. If the remote peer is not allowed to access the invoked peer, the INVOCATION INTERCEPTOR stops the invocation and sends a REMOTING ERROR to the client, otherwise access is granted. INVOCATION INTERCEPTORS are triggered by callbacks (modeled using the Callback primitive), as can be seen in Fig. 26. Naturally the interceptors on the client-side and the server-side are linked through a Virtual Connector.

Often interceptors for one and the same task exist both on client-side and server-side. In Fig. 27 three examples are presented. Logging is needed both on client-side and server-side, but no Virtual Connector between the logging interceptors is necessary. The server-side federation interceptor checks whether an invoking peer belongs to a federation or not. The client-side federation interceptors thus must put the federation information of the invoking peer into the INVOCATION CONTEXT. Thus there is a virtual communication between these two interceptors, which is modeled using a Virtual Connector. Likewise, the client-side and server-side authentication interceptors need to transmit authentication information over the wire.

6. A model validator for the architectural primitives

To further support the use of the architectural primitives in model-driven software development, we have developed a model validator, which can be used as a plug-in in a model-driven tool chain (such as the OpenArchitectureWare generator [33]). The plug-in is capable of validating architectural models that conform to UML2 meta-models (like the excerpt in Fig. 1 or other compliant meta-models)

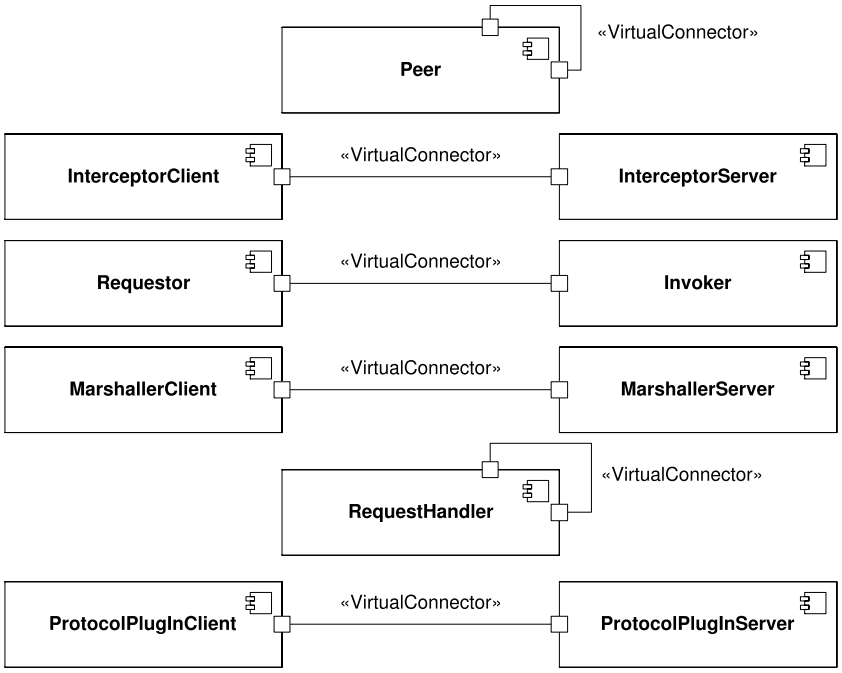

Fig. 24. Virtual communication among Leela components.

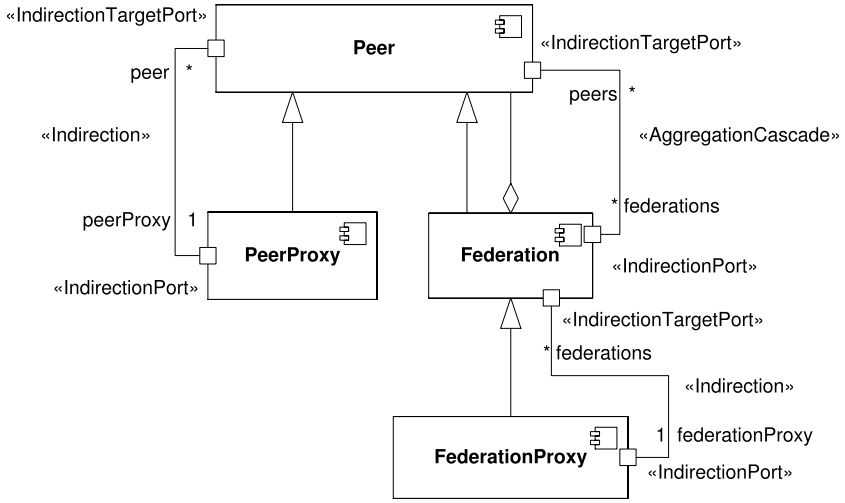

Fig. 25. Proxy-based indirection in Leela.

and OCL constraints. We have specified the proposed architectural primitives UML profile using the tool, in order to validate architectural models that contain such primitives. Based on both the UML meta-model and the primitives profile, the validator can parse architectural models and check that the constraints of the primitives are not violated. The models produced by the validator can be then used as input for code generators.

We use the language Frag [49,48] as the syntactic foundation for defining the UML meta-model, the architectural primitives profile, as well as the UML models per se. Frag's main goal is to provide a tailorable language. Among other things, Frag supports the tailoring of its object system and the extension with new language elements. In addition to the UML2 meta-model and the meta–meta-model, we have defined a constraint language which follows the OCL constructs.

The process of using the plug-in conforms to a typical workflow for model-driven development. First, the input models need to be read and transformed into the special Frag syntax for UML2 models and the architectural primitives, which are defined using a meta-model and a profile, respectively. The UML2 models can either be written with UML tools (with XMI export) or directly in the textual Frag syntax. If a UML tool is used, the XMI export is transformed into the textual Frag syntax. Second, the application models get validated with respect to the UML meta-model, as well as the OCL constraints defined

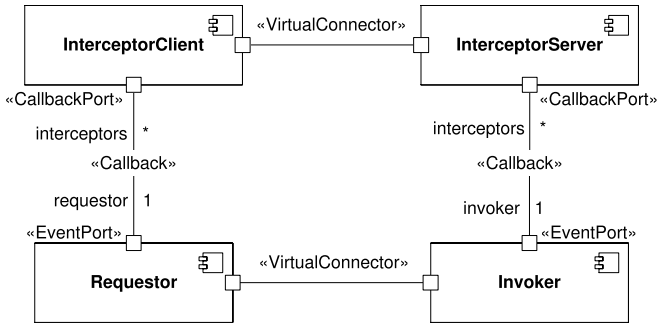

Fig. 26. Invocation interceptors in the invocation chain.

upon them with respect to the primitives profile. After the model is validated it is transformed into an EMF model, which is understood by the code generator. The latter creates the code in the target output languages, such as Java. This tool chain is depicted in Fig. 28. In between this sequence, other steps may be inserted, such as domain-specific model transformations.

In our plug-in, we define all meta-models, including the UML meta-model, on top of one common meta–meta-model. The meta–meta-model can be very simple, or more elaborate like the OMG Meta Object Facility (see http://www.omg.org/mof/). The meta–meta-model is used to define meta-models. In addition, the constraint language is defined using the meta–meta-model, with which models can be constrained at the meta-level and hence validated at the model level. In the case of our plug-in and the Frag language, we have defined a simple meta–meta-model that reuses Frag's language features wherever possible. An excerpt of this meta–meta-model for defining UML2 meta-models is shown in Fig. 29. The meta–meta-model is based on the most general class in the Frag object system: `Object`. The meta–meta-model classes are sub-classes of `ConstrainedClass` which allows to add OCL-style constraints to classes. The convenience class `ConstraintChecker` looks up all `ConstrainedClass` instances via reflection and checks the constraints. Constraints are specified in a language similar to OCL (defined using the class `FCL`). The meta-models are defined using `Class`. We introduce also a number of relationships between classes: Dependencies, Associations, Compositions, and Aggregations. In addition, typed attributes can be specified. Please note that we do not define the generalization relationship, because multiple inheritance is suitably predefined by Frag and we can reuse this implementation. The `Stereotype` class defines the UML2 extends-relationship; that is, it allows to extend metaclasses. `Enum` is a convenience class to define Enumeration types.

Using the meta–meta-model, we can define meta-models like the UML2 meta-model shown in Fig. 1. As an example, consider the Component and Namespace metaclasses and two associations of Component of the UML2 meta-model in the Frag syntax:

```
namespace eval UML2 {
  ...
  MMM::Class create Component \
  -superclasses \
  {PackageableElement Class}
  MMM::Association create \
  ComponentInterfaceRequired -ends {
    {Component -multiplicity *
     -navigable 0}
    {Interface -roleName required
     -multiplicity *
     -navigable 1}
  }
  MMM::Association create \
  ComponentInterfaceProvided -ends {
```

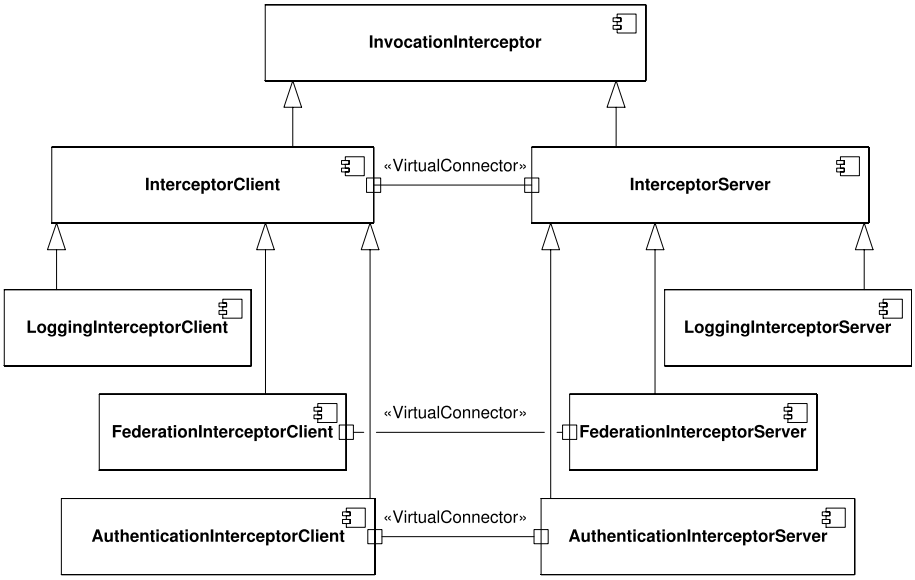

Fig. 27. Special invocation interceptors.

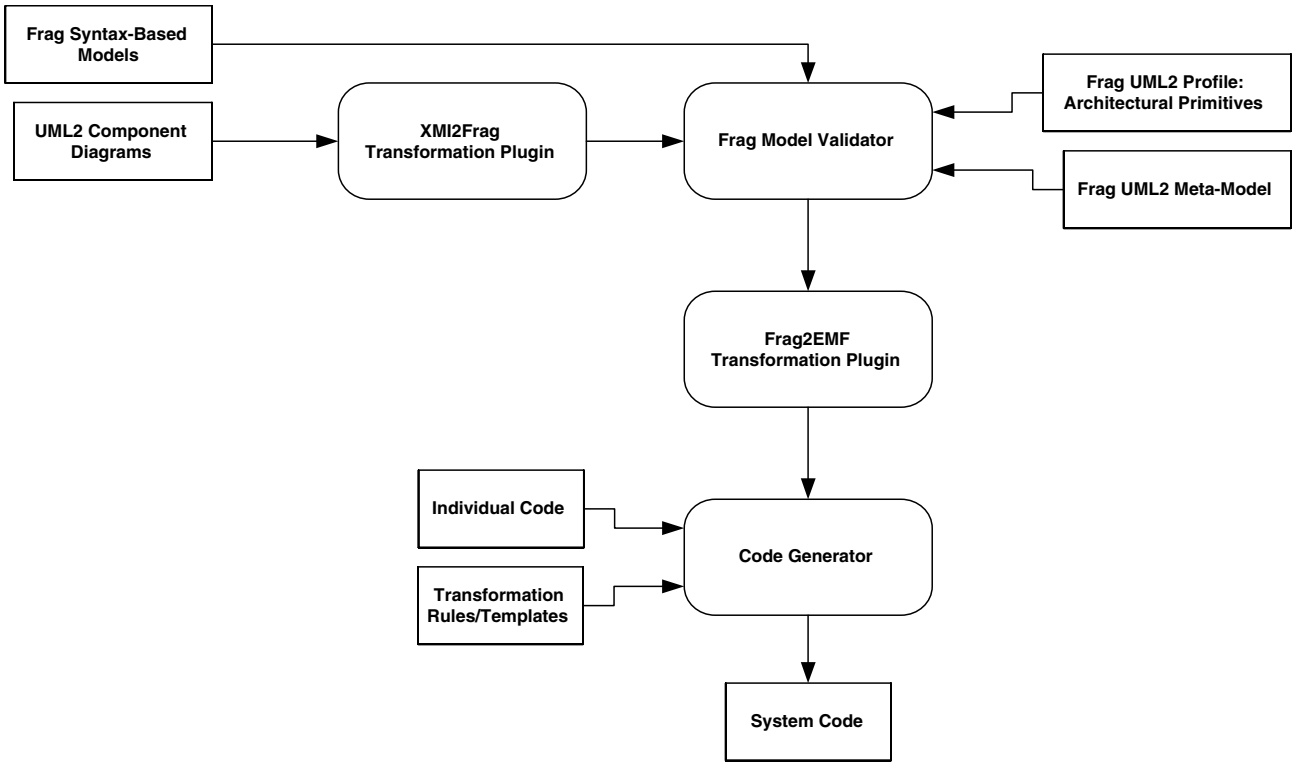

Fig. 28. Tool Chain Overview.

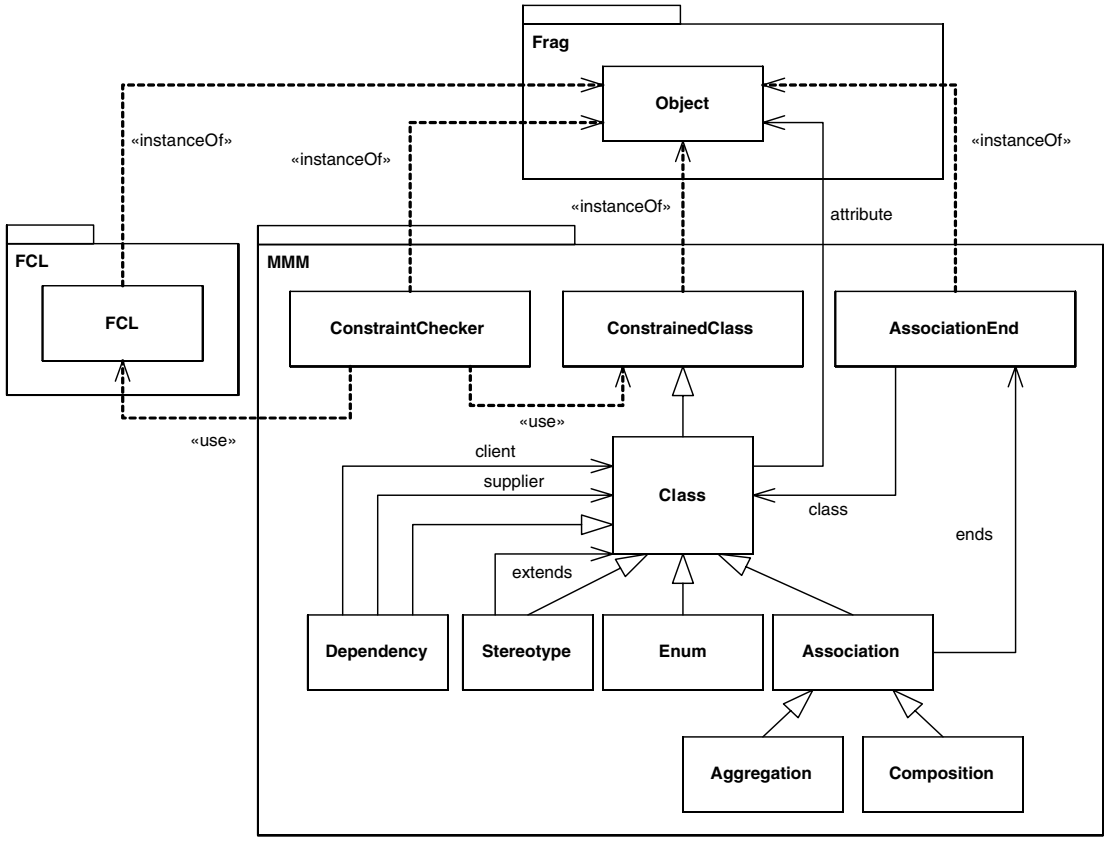

Fig. 29. Validator Tool UML2 meta–meta-model – excerpt.

```
    {Component -multiplicity *
     -navigable 0}
    {Interface -roleName provided
     -multiplicity *
     -navigable 1}
  }
  MMM::Class create Namespace \
  -superclassesNamedElement
  ...
}
```

Once the UML meta-model is defined, the UML profile for the architectural primitives needs to be specified. Consider, as an example the textual definition of the Grouping primitive. Again, we can also use the model transformers to generate this textual model from graphical models:

```
namespace eval Grouping {
  MMM::Stereotype create Group \
  -extendsUML2::Package


  Group addInvariant {
    [FCL size [[self basePackage] \
    ownedMember]]== 0
  }
  Group addInvariant {
    [FCL forAll im [[self basePackage] \
    importedMember] {
      [FCL isKindOf $im UML2::Component]
    }]
  }
}
```

The primitive specification first defines the necessary stereotype for Grouping, and then it provides all constraints required for that primitive. It is noted that the constraints are almost a one-to-one translation of the OCL constraints explained before for Grouping, but in Frag constraint syntax as in the example above (called FCL). We can use the OCL-to-FCL transformer to automatically generate FCL code from OCL, and vice versa. The UML profile for the architectural primitives is thus translated into a set of such specifications in the Frag syntax, and subsequently used to parse and validate models.

As a small example, we consider a small model with the following elements (also shown graphically in Fig. 30):

```
UML2::Component create WorkflowEngine
UML2::Component create ProcessIntegrationAdapter
UML2::Component create Dispatcher


UML2::Package create WorkflowCorrelationGroup \
-importedMember {
  WorkflowEngine ProcessIntegrationAdapter
  Dispatcher
}


Grouping::Group create gl \
  -basePackage WorkflowCorrelationGroup
```

In this example model, we first specify three UML components: a workflow engine, a process integration adapter, and a dispatcher. Then we define a Package that depicts a correlation group of three components and we use the Group stereotype to apply the relevant constraints of the Grouping architectural primitive. Our model validator automatically checks all constraints, once a model is assembled. In this example, no problems are found, but if we would for instance have owned members in the Package or other member types then Components, the model validation would fail.

It is noted that our syntax introduced above is not necessarily intended to be used by developers. Rather it can itself be generated from the export of an UML2+OCL tool. That is, the developers can also specify UML2 models and constraints graphically, and the models can then be validated in the model-driven tool chain. During this validation, the primitives are automatically checked as well.

7. Lessons learned

Our case study in Section 5 was conducted to demonstrate that a non-trivial system, that was initially modeled and implemented independently of our approach can retroactively be modeled using our pattern primitives. During the application of the approach and the application of the model validator, presented in Section 6, a number of inconsistencies in the initial models have been found and

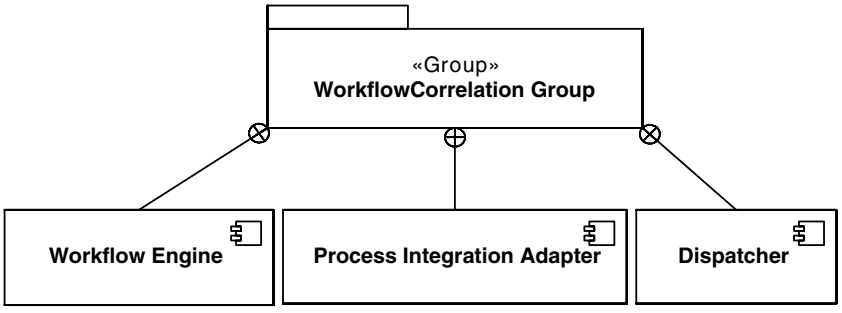

Fig. 30. Example model for workflow correlation.

corrected. This is an important benefit, as the proposed approach can help to improve the quality of a system's current documentation. It also demonstrated that each of the primitives were variable enough to deal with multiple situations, where the respective patterns were applicable.

Furthermore, we have used the resulting models in a number of student projects. Despite the fact that a higher quantity of information was given through the stereotypes, the students did not find the enriched primitives models less intuitive than the original models. In contrast – after some initial explanation – the primitives conveyed the design intent of our case study prototype in a better way. This proved especially useful for evolution of the prototype: changes have been made without violating the design constraints of the patterns. This aspect is especially important, when systems are generated from the models using a model-driven approach. Also, the students deemed that they were able to link the primitives to the patterns, which document important forces and consequences of the design decisions (especially the consequences of architectural patterns to the system's quality attribute). This link, however, could be improved through better tool support, specifically targeted at making this link explicit.

When modeling the case study we have mainly used textual representations of the models, as shown in Section 6, and created the graphical UML representation based on them. For the model validator, we have also realized an integration with EMF, to demonstrate the interplay with existing modeling tools. This, however, often requires customizations of the models or tools, as there are usually certain differences between the various UML tools.

We have selected UML, a de-facto standard modeling language in software architecture, in order to guarantee broad tool support and familiarity of modelers with the language. However, the main shortcoming of this approach stems from the very own use of UML. The extension mechanisms of UML, in particular the stereotypes, are cumbersome to use because of their second-class status: they are neither metaclasses of the standard meta-model, nor model elements and this fact often confuses the users of UML. Furthermore, OCL constraints, even though they provide semi-formal semantics to the stereotypes, are not well accepted in the software architecture community, partly because there are no tools so far that can dynamically check the constraints in UML models. Lastly, the constant evolution of the UML standard, forces us to update the mapping of the architectural primitives in the language in its subsequent versions, which can prove to be cumbersome. However, we do believe that the advantages that UML conveys outweigh these disadvantages.

In addition, our choice to use only UML profiles limits the capabilities of the visual representations of the models. In particular, as can be seen in the case study, the UML representations may quickly lead to visual cluttering, especially if multiple primitives are combined in one model. One solution would be to build customized UML tools that can lessen the visual clutter, e.g. by visually replacing some of the primitives' stereotypes with textual notes. This way, the primitives are still formalized in the model, but the graphical representation is more usable. Another solution is to use other modeling environments, which introduce their own abstractions as new modeling language elements, and can thus provide richer visualizations. For instance, the Generic Modeling Environment (GME) [24] provides a notion of hierarchy in its modeling syntax, which is a clearer representation of the concern represented by aggregation and composition cascade than the UML representation. In this research, we deliberately focused only on UML profiles, for the reasons explained before, but as future work we plan to explore the possibilities of using domain-specific modeling languages as concrete syntax and our UML profiles as abstract syntax (e.g. following the approach described in [15]) to combine the best of both worlds.

The primitives that we have proposed are structural in nature, as they concern the composition of components and connectors as well as interface matching between them. Nonetheless, architectural patterns contain also a behavioral part that mandates the interaction of architectural elements, which has not been studied yet. For example The Callback primitives can be used to compose the Model-View-Controller, but the interaction protocol between the Model on the one hand and Views and Controllers on the other hand, is not covered by the current definition of the primitive. We aim to extend this work with behavioral models for the individual primitives, as mentioned in the future work section. We have already performed this in a another approach, where we have successfully integrated activity diagrams and class diagrams via OCL constraints for modeling compositions in dynamic programming environments (see [50]). Even though this is quite a different modeling problem, the general approach for integrating activity diagrams with structural models can be followed for the primitives approach reported in this paper as well.

The proposed primitives are a modeling solution that can address the inherent variability of patterns – an issue that makes the modeling of the pattern participants themselves highly problematic. However, this does not automatically make the patterns explicit in an architectural design – the primitives only give a hint of the application of specific patterns. An architect needs to further annotate a specific collaboration of primitives to denote their synergy and implementation of the patterns semantics.

The full benefits of our approach can be obtained when full tool support is accessible to ordinary developers. We have started working on model-driven tool support by providing a model validator (see Section 6), which is an important aid for composing and decomposing primitive models via model-driven development. In addition, further visualizations of the composition/decomposition would be helpful. Also tool support is needed to facilitate the composition of primitives into patterns, making explicit that a set of primitives with customized constraints form a specific pattern variant. This would furthermore support

analysis of the design, by linking each pattern with the quality attributes it affects positively or negatively.

8. Related work

The approach described in this paper is based on related research on architectural primitives, UML profiles for architectural description, and modeling architectural patterns. Table 1 gives an overview of the related work, and how it compares to the approach presented in this paper.

The idea of primitives as the fundamental elements of architectural patterns and design patterns has been investigated from several viewpoints. Pree has worked in the area of object-oriented frameworks and has explored primitives in the construction of 'hot spots', i.e., variation points that are adapted in individual applications [35,36]. His primitives are defined in two levels of abstraction: At a lower level, the fundamental elements of patterns are 'hook' and 'template' methods and their corresponding classes; at a higher level the aforementioned fundamental elements are used to specify composition patterns for hot spots that are called *meta-patterns*. These composition patterns themselves can be used for specifying even higher-level patterns, such as the GoF [11] patterns; however they are not architectural elements and thus cannot be used to describe architectural patterns like the architectural primitives in this paper.

In the realm of patterns, many patterns are described as compound patterns that consist of other, existing patterns. For instance, in [44] the BROKER pattern is described as a compound pattern composed from patterns from [44,39,11,6]. Our approach follows a similar philosophy as we define primitives that can be used to model architectural patterns, but is different in that these architectural primitives are more specific and formally specified than patterns. The primitives can be seen as participants of patterns, whereas patterns require substantial hand-crafting (i.e., a design and implementation effort) in order to be used as part of another pattern.

Mehta and Medvidovic proposed a framework, called Alfa, for composing architectural patterns through *architectural primitives* [28] that are certain underlying concepts, shared by all patterns. They propose a number of such primitives as the building blocks for constructing the architectural elements of patterns and demonstrate their approach through the representation of several architectural patterns through the primitives. This approach is based on the assumption that there exists a fixed set of fundamental primitives that can potentially characterize any architectural pattern participant and therefore this framework of primitives can be used for characterizing and comparing patterns. Our approach is different in the sense that we investigate architectural primitives at a larger granularity and level of abstraction. Moreover, our primitives are recurring concepts in several, but not all, architectural patterns, and they are characterized by rich semantics that serve specialized purposes.

Similarly, Bass et al. in the first edition of [3] had proposed a predefined set of *unit operations*, such as separation, abstraction, compression and resource sharing as the building blocks for all architectural and design patterns. In contrast to our architectural primitives, these unit operations are defined at a much higher level of abstraction. They rather describe atomic architectural transformations and operations, whereas our primitives describe fundamental, recurring structures. The same authors, in the second edition of their book propose a number of *architectural tactics* for controlling the response of quality attributes. Architectural patterns packages a number of tactics, in the sense that the consequences of applying the pattern is the realization of one or more tactics. Tactics are abstract hints on how to support a specific quality attribute and are not directly related to how an architectural pattern is modeled.

There have also been several attempts for specifying existing ADLs or proposing new ADLs as extensions of UML, usually in the form of profiles. Medvidovic et al. have pointed out three different ways to use UML as an ADL [26]: (a) using the "pure" UML meta-model "as

Table 1
Overview and comparison of related work

Approach	Building blocks	Granularity	Application	Semantics
Hot spots [35,36]	Class methods	Class	Object-oriented frameworks	Object-oriented
Alfa primitives [28]	Form and function of architectural patterns	Component and connector	Architectural patterns	General characteristics
Unit operations [3]	Abstract principles	Component	Architecture	informal
Architectural tactics [3,7,13,20,4,40]	Abstract principles or heuristics	Component and connector	Quality-attribute driven design	Informal
UML profiles [26]	UML stereotypes	Components and connectors	Architectural description	ADLs
Acme [14]	Templates for patterns	Components and connectors	Architectural description	Acme ADL
Formal approaches to modeling patterns [8,29,43,25]	Language constructs	Class	Design patterns	Formalization of one specific pattern variant
Primitives of architectural patterns	Participants of architectural patterns	Component and connector	Architectural patterns	Specialized and formalized

is'', which forces the architect to implicitly define the necessary architectural concepts; (b) constraining the UML meta-model through profiles and thus providing explicitly the architectural concepts as constrained stereotypes, while still conforming to the standard meta-model; (c) modifying the UML meta-model and thus providing ''native'' support for architectural description, but losing conformance to the standard meta-model. They have also evaluated the first two approaches by using them to map three ADLs to UML.

An ADL that treats architectural patterns as first-class entities is Acme [14] supported by the AcmeStudio tool [38]. The language itself provides built-in templates that can be used to model patterns, while AcmeStudio has some well-established patterns (e.g. Layers, Pipes and Filters, Client–Server) in its default package. However, the syntax support offered by Acme is rather limited, as it provides a fixed set of architectural elements like components, connectors, ports, roles etc. Our approach aims at more flexibility by providing a wider range of lower-level primitives, such as namespaces or aggregations used in Grouping and Aggregation Cascade.

Clements et al. in [7] demonstrated how UML 1.x can be used ''as is'' in representing the fundamental architectural concepts in a number of architectural views. This work was continued by Garlan et al. in [13] and later by Ivers et al. in [20] to take under account the forthcoming UML 2.0, and particularly focus on the provision for the component and connector view in the new standard. The improvements of the new UML 2.0 meta-model for architectural concepts, especially ports and internal structures, was also advocated by Björkander and Kobryn in [4]. Finally Selic and Rumbaugh [40,41] have defined a UML profile for real-time systems, UML-RT, which embodies several architectural concepts such as components (so-called ''capsules''), connectors, and ports. Our approach uses a different line of attack: we do not model the architectural concepts that are specific to an architectural pattern, but rather the fundamental primitives that participate in a number of patterns. Thus we overcome the limitations of ADLs by providing a wealth of abstractions, capable of modeling several of the well-known architectural patterns.

There are many approaches for modeling or representing software patterns, the vast majority of which focuses on the design patterns from [11]. A number of such approaches attempted to formally specify these patterns (see for instance [8,29,43,25]). These approaches, however, have not gained much momentum in recent years mainly because of their complexity and their resulting limitations regarding their practical use. Moreover, these approaches have not been used for architectural patterns or whole pattern languages, like our primitives, but just for some isolated patterns from [11]. A third major difference of these approaches, compared to our approach, is that they only support one variant of a pattern (often simply following the C++ example from [11]) and not other possible pattern variations. The same problem appears also when using the Collaboration metaclass provided by UML 2.0 to describe a design pattern. Most patterns (especially architectural patterns), however, can be realized using a multitude of different design variants. Our approach describes primitives that are participants of the patterns and can be tailored to support multiple variants of a pattern. In other words, we can model the variants of a pattern, by constraining the specific semantics of the architectural primitives that comprise the pattern.

There have also been some approaches that propose language support for design patterns, such as [31,5], or implementations of patterns as aspects, such as [16,17]. These approaches make patterns first-class entities of the language or aspect framework, and thus they become more traceable in the code than a pattern implementation scattered across a number of classes. All of these approaches might be considered as a way to better understand the use of a single pattern in an architecture, but not for documenting the design of complex architectures based on (multiple) patterns, as this paper advocates.

9. Conclusions and future work

We have proposed modeling architectural patterns through a number of architectural primitives in the component and connector view. We have elicited an initial set of these primitives from a pool of established architectural patterns in order to ensure their correctness and broad applicability. This set of primitives is original and helps solving the fundamental problems in modeling architectural patterns that were outlined in Section 1: they offer the necessary abstractions that grasp the rich semantics found in patterns; they can represent not only a specific pattern variant but multiple variants of a pattern, by tailoring the architectural primitives with constraints. We have validated our approach by modeling the primitives in the well-known Eclipse/Octopus tool set, applying it to a number of case studies (one of them was presented in Section 5), and developing our own model validator prototype to support model-driven development using our concepts (see Section 6 for details).

We plan to extend this work in the following directions:

- document the architectural primitives of other domain-specific patterns and pattern languages in the component and connector view;
- provide the explicit modeling of patterns through the collaboration of a group of primitives by annotating them and adding semantics to them;
- experiment on modeling the variability of the patterns, not only by modifying the constraints of primitives but also by combining alternative primitives in given patterns;

- relate our approach to the notion of pattern languages (larger collections of interrelated patterns). In particular, we plan to document more patterns from the remoting pattern language (see [44]) and a pattern language for general-purpose architectural patterns (see [2]);
- offer support for a better visualization (or concrete syntax) of the primitives than the current representation as UML stereotypes;
- add behavioral modeling (e.g. based on activity or sequence diagrams or state machines) to the pattern primitives;
- search for architectural primitives in other views, such as the module view;
- offer the validation tool as a ready-to-use Eclipse plug-in that can be used in cooperation with other model-driven development plug-ins.

References

[1] P. Avgeriou, N. Medvidovic, N. Guelfi, Software architecture description with UML, in: J. Nunes, B. Selic, A. Silva, A. Toval (Eds.), UML Modeling Languages and Applications – UML 2004 Satellite Activities, LNCS, 3297, Springer Verlag, Lisbon, Portugal, 2004.

[2] P. Avgeriou, U. Zdun, Architectural patterns revisited – a pattern language, in: Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, July 2005.

[3] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, first (second) ed., Addison Wesley, Reading, MA, USA, 1998, 2003.

[4] M. Björkander, C. Kobryn, Architecting systems with UML 2.0, IEEE Softw. 20 (4) (2003) 57–61.

[5] J. Bosch, Design patterns as language constructs, J. Object Orient. Program. 11 (2) (1998) 18–32.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture – A System of Patterns, John Wiley and Sons Ltd, 1996.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002.

[8] A.H. Eden, Y. Hirshfeld, LePUS – symbolic logic modeling of object oriented architectures: A case study, in: Second Nordic Workshop on Software Architecture – NOSA'99, Ronneby, Sweden, April, 1999.

[9] T. Foster, L. Zhao, Cascade, J. Object Orient. Program. 11 (9) (1999).

[10] M. Fowler, Analysis Patterns, Addison-Wesley, 1997.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[12] J. Garcia-Martin, M. Sutil-Martin, Virtual machines and abstract compilers – towards a compiler pattern language, in: Proceedings of EuroPlop 2000, Irsee, Germany, July, 2000, pp. 375–396.

[13] D. Garlan, S.-W. Cheng, A.J. Kompanek, Reconciling the needs of architectural description with object-modeling notations, Sci. Comput. Program. 44 (1) (2002) 23–49.

[14] D. Garlan, R. Monroe, D. Wile, Acme: An architecture description interchange language, in: CASCON'97, in: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, 1997, p. 7.

[15] J. Greenfield, K. Short, Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools, John Wiley and Sons Ltd, 2004.

[16] J. Hannemann, G. Kiczales, Design pattern implementation in Java and AspectJ, in: C. Norris, J.J.B. Fenwick (Eds.), Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02), ACM SIGPLAN Notices, 37, 11, ACM Press, New York, 2002, pp. 161–173.

[17] R. Hirschfeld, R. Lämmel, M. Wagner, Design patterns and aspects – Modular designs with seamless run-time integration, in: Proc. of the 3rd German GI Workshop on Aspect-Oriented Software Development, Technical Report, University of Essen, 2003, p. 8.

[18] C. Hofmeister, R. Nord, D. Soni, Applied Software Architecture, Addison-Wesley Longman Publishing Co., Inc., 2000.

[19] IEEE. Recommended Practice for Architectural Description of Software Intensive Systems. Technical Report IEEE-std-1471-2000, IEEE, 2000.

[20] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, J.R.O. Silva, Documenting component and connector views with uml 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, 2004.

[21] R. Johnson, B. Woolf, Type object, in: R.C. Martin, D. Riehle, F. Buschmann (Eds.), Pattern Languages of Program Design 3, Addison-Wesley, 1998.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of ECOOP'97, LCNS 1241, Springer-Verlag, Finnland, 1997.

[23] P. Kruchten, The 4+1 view model of architecture, IEEE Softw. 12 (6) (1995) 42–50.

[24] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi, The Generic Modeling Environment, in: Workshop on Intelligent Signal Processing, vol. 17, Budapest, Hungary, May, 2001.

[25] J.K.H. Mak, C.S.T. Choy, D.P.K. Lun, Precise modeling of design patterns in uml, in: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, 2004, pp. 252–261.

[26] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, J.E. Robbins, Modeling software architectures in the unified modeling language, ACM Trans. Softw. Eng. Methodol. 11 (1) (2002) 2–57.

[27] N. Medvidovic, R.N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Trans. Softw. Eng. 26 (1) (2000) 70–93.

[28] N.R. Mehta, N. Medvidovic, Composing architectural styles from architectural primitives, in: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, Helsinki, Finland, 2003, pp. 347–350.

[29] T. Mikkonen, Formalizing design patterns, in: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, Kyoto, 1998, pp. 115–124.

[30] R.T. Monroe, D. Garlan, Style-based reuse for software architectures, in: Proceedings of the Fourth International Conference on Software Reuse, 1996.

[31] G. Neumann, U. Zdun, Filters as a language support for design patterns in object-oriented scripting languages, in: Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems, San Diego, California, USA, May, 1999.

[32] OMG. UML 2.0 superstructure final adopted specification, Technical Report ptc/03-08-02, Object Management Group, August, 2003.

[33] Open Architecture Ware. openArchitectureWare 4.1. http://www.openarchitectureware.org/, 2006.

[34] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Softw. Eng. Notes 17 (4) (1992).

[35] W. Pree, Metapatterns: a means for capturing the essentials of object-oriented design, in: Eur. Conf. Object Orient. Program. (ECOOP), Springer-Verlag, Bologna, 1994, pp. 4–8.

[36] W. Pree, Hot-spot-driven framework development, in: R.J.M. Fayad, D. Schmidt (Eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley & Sons, 2000.

[37] J.E. Robbins, N. Medvidovic, D.F. Redmiles, D.S. Rosenblum, Integrating architecture description languages with a standard design

method, in: Proceedings of the 20th ICSE, IEEE Computer Society, Kyoto, 1998, pp. 209–218.
[38] B. Schmerl, D. Garlan, Acmestudio: Supporting style-centered architecture development (research demonstration), in: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, 2004, pp. 23–28.
[39] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Patterns for Concurrent and Distributed ObjectsPattern-Oriented Software Architecture, John Wiley and Sons Ltd, 2000.
[40] B. Selic, Turning clockwise: using UML in the real-time domain, Commun. ACM 42 (10) (1999) 46–54.
[41] B. Selic, J. Rumbaugh, Using UML for modeling complex real-time systems, 1998.
[42] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Addison-Wesley, 1996.
[43] N. Soundarajan, J.O. Hallstrom, Responsibilities and rewards: Specifying design patterns, in: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, 2004, pp. 666–675.
[44] M. Voelter, M. Kircher, U. Zdun, Remoting PatternsPattern Series, John Wiley and Sons, 2004.
[45] U. Zdun, Patterns of tracing software structures and dependencies, in: Proceedings of EuroPlop 2003, Irsee, Germany, June, 2003.
[46] U. Zdun, Loosely coupled web services in remote object federations, in: Proceedings of the Fourth International Conference on Web Engineering (ICWE'04), Munich, Germany, July, 2004.
[47] U. Zdun, Some patterns of component and language integration, in: Proceedings of the 9th European Conference on Pattern Languages of Programs (EuroPlop 2004), Irsee, Germany, July, 2004.
[48] U. Zdun. Frag. <http://frag.sourceforge.net/>, 2005.
[49] U. Zdun, Tailorable language for behavioral composition and configuration of software components, Comput. Lang. Syst. Struct. 32 (1) (2006) 56–82.
[50] U. Zdun, M. Strembeck, Modeling composition in dynamic programming environments with model transformations, in: Proc. of the 5th International Symposium on Software CompositionLNCS 4089, Springer-Verlag, Vienna, Austria, 2006.